Spark Streaming 技术内幕及源码剖析

王家林 夏 阳 编著

- 彻底解密Spark Streaming**架构设计**
- 深入挖掘Spark Streaming**运行机制**
- 逐行剖析Spark Streaming**框架源码**
- 手把手讲解Spark Streaming**性能调优**

清華大學出版社

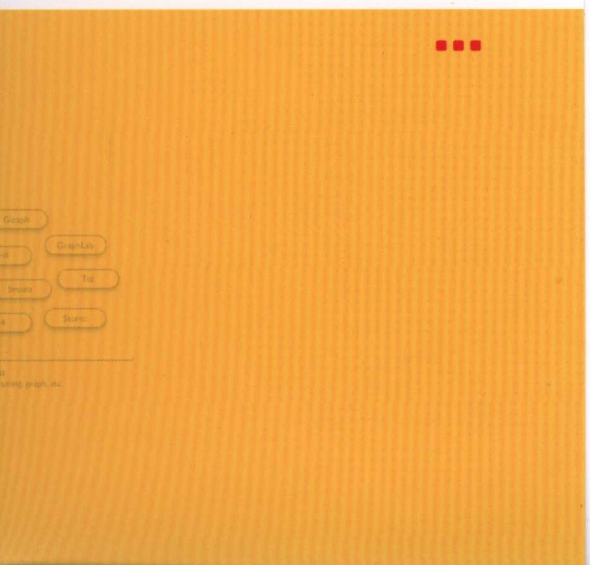
■ 作者简介

王家林

中国著名的Spark培训专家，Apache Spark、Android技术中国区布道师，DT大数据梦工厂创始人和首席专家，Android软硬件整合专家。深入研究了Spark从0.5.0到2.1.0中共28个版本的Spark源码，目前致力于开发优化的Spark中国版本。尤其擅长Spark在生产环境下各种类型和场景故障的排除和解决，痴迷于Spark在生产环境下任意类型(例如Shuffle和各种内存问题及数据倾斜问题等)的深度性能优化。

夏阳

系统架构师，从事平台和应用软件研发工作多年，行业阅历丰富，对行业技术发展有独到见解和精准判断，曾就职于中创中间件公司、蚁坊软件公司、任子行网络技术股份有限公司。对大数据处理、机器学习、图计算、文本处理等技术领域有丰富的实战经验和浓厚兴趣。



Spark Streaming

技术内幕及源码剖析

王家林 夏 阳 编著

清華大學出版社
北京

内 容 简 介

本书以大数据处理引擎 Spark 的稳定版本 1.6.x 为基础，从应用案例、原理、源码、流程、调优等多个角度剖析 Spark 上的实时计算框架 Spark Streaming。在勾勒出 Spark Streaming 架构轮廓的基础上，从基本源码开始进行剖析，由浅入深地引导已具有 Spark 和 Spark Streaming 基础技术知识的读者进行 Spark Streaming 的进阶学习，理解 Spark Streaming 的原理和运行机制，为流数据处理的决策和应用提供了技术参考；结合 Spark Streaming 的深入应用的需要，对 Spark Streaming 的性能调优进行了分析，也对 Spark Streaming 功能的改造和扩展提供了指导。

本书适合大数据领域 CTO、架构师、高级软件工程师，尤其是 Spark 领域已有 Spark Streaming 基础知识的从业人员阅读，也可供需要深入学习 Spark、Spark Streaming 的高校研究生和高年级本科生参考。

图书在版编目(CIP)数据

Spark Streaming技术内幕及源码剖析 / 王家林，夏阳编著. — 北京：清华大学出版社，2017
ISBN 978-7-302-46491-4

Ⅰ. ①S… Ⅱ. ①王… ②夏… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字（2017）第 025588 号

责任编辑： 袁金敏 战晓雷
封面设计： 刘新新
责任校对： 徐俊伟
责任印制： 李红英

出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 清华大学印刷厂
经　　销： 全国新华书店
开　　本： 185mm×230mm　　**印　　张：** 16.25　　**字　　数：** 264 千字
版　　次： 2017 年 5 月第 1 版　　**印　　次：** 2017 年 5 月第 1 次印刷
印　　数： 1 ～ 3000
定　　价： 49.00 元

产品编号：072312-01

前 言

大数据浪潮汹涌来袭，这绝不仅仅是信息技术领域的革命，更是在全球范围引领社会变革的机遇。大数据的集群计算开源软件Spark在大数据计算平台应用领域日益凸显其重要地位。如果大数据技术领域从业人员的技术水平仍停留在只知使用开源软件，而不从开源软件的原理、架构上去理解，不到源码中去体会细节，则难以从根本上彻底解决现实工作中遇到的技术问题，更难以胜任大数据领域的技术创新工作。

可以预见，大数据的处理将越来越强调实时处理。Spark Streaming是建立在Spark上的实时计算框架，在Spark的各子框架中处于举足轻重的地位。彻底掌握 Spark Streaming的同时，也能加深对Spark Core技术的理解和掌握，还能具备开发同样高端的Spark应用程序的实力。对于有志向的Spark学习进阶者来说，深入了解Spark Streaming的源码是提高核心竞争力的捷径。

本书不仅对Spark Streaming的API做总结性介绍，而且重点针对Spark 1.6.x的Spark Streaming进行源码剖析。该书的开始部分对Spark的基本原理有一些阐述，但主要是彻底深入剖析Spark Streaming的内部原理。

读源码的人都怕自己走进大量源码的迷宫。为了提高源码学习效率，本书在剖析源码前，会对源码实现的功能的大致原理和流程轮廓进行介绍。书中有方便源码剖析的流程图，这对于理解和掌握Spark Streaming的各个功能非常重要。读者看到复杂的流程图时不一定要立刻全部理解掌握，但可以在源码学习过程中经常回过头来对照流程图以加深印象。

为了在书的页面内清晰展示复杂的流程图，书中绝大多数流程图采取了从上至下的树状结构来体现调用关系。每个方框中注明了类和方法，被其调用的类的方法会在下一行从左至右依次显示，调用和被调用的类方法间用有向线连接。有些方框上部会给出类的成员

变量，其类型就是方框中指明的类。粗箭头不是表示调用关系，而是表示传递消息。

源码剖析过程中，源码中关键的类名、方法名、注释会以粗体显示，使读者清楚重点。读者应以粗体部分为重点进行阅读，其他部分可以粗略浏览。有些源码篇幅过大，可能会省略其中的部分代码，以突出当前读者需要阅读的源码主体。

王家林　夏　阳

2017年2月27日于北京

目 录

第1章
Spark Streaming应用概述

Spark是一个类似于Hadoop的MapReduce的分布式计算框架，其核心是弹性分布式数据集（Resilient Distributed Dataset，RDD），提供了比MapReduce更丰富的模型，可以在内存中快速对数据集进行多次迭代，以支持复杂的数据挖掘算法和图形计算算法。

Spark包含Spark Core、Spark SQL、Spark Streaming、MLlib、GraphX等主要部件，如图1-1所示。

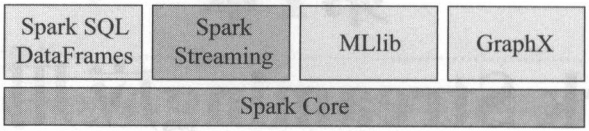


图1-1 Spark的主要部件

其中的Spark Core包含Spark的基本功能，是核心的计算引擎，并定义了RDD的API。其他的Spark部件都是构建在Spark Core之上的。

Spark Streaming是建立在Spark上的实时计算框架，提供了丰富的API，以支持Spark处理大规模流式数据。

这是一个流处理的时代。Spark十分强大的一个重要原因在于，它的流式处理可以在线使用图计算、机器学习或者SparkR的成果，这得益于Spark一体化、多元化的基础架构设计。在Spark Streaming中可以调用Spark SQL、MLlib、GraphX等其他子框架，无须任何设置。这是Spark无可匹敌之处。但Spark的应用中，Spark Streaming也是很容易出问题的。

Spark Streaming与其他子框架不同之处在于，它更像是Spark Core之上的一个复杂应用程序。如果要做Spark的定制开发，Spark Streaming的剖析也会提供最好的参考。

1.1 Spark Streaming应用案例

Spark Streaming应用程序运行的时候，往往在短时间内会产生大量日志信息，不利于研究分析。可以通过加大批处理时间间隔（batch interval）来降低批处理频率，减少日志

信息量，以便看清楚各个环节。

下面从一个Spark Streaming应用程序的开发入手，观察运行过程，以增强感性认识。以下是一个广告点击的在线黑名单过滤的Spark Streaming应用程序，程序中有详细注释，以方便初次接触Spark Streaming的读者理解。

源码1-1　OnlineBlackListFilter

```
package com.dt.spark.streaming


import org.apache.spark.SparkConf
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds


object OnlineBlackListFilter {
    def main(args: Array[String]){
      /**
       * 第1步：创建Spark的配置对象SparkConf，设置Spark程序的运行时的配置信息，
       * 例如，通过setMaster来设置程序要链接的Spark集群的Master的URL，如果设置
       * 为local，则代表Spark程序在本地运行，特别适合于机器配置条件非常差（例如
       * 只有1GB内存）的初学者
       */
      // 创建SparkConf对象
      val conf = new SparkConf()
      // 设置应用程序的名称，在程序运行的监控界面可以看到名称
      conf.setAppName("OnlineBlackListFilter")
      // 此时，程序在Spark集群
      conf.setMaster("spark://Master:7077")
```

```
val ssc = new StreamingContext(conf, Seconds(30))

/**
 * 黑名单数据准备。实际上黑名单一般都是动态的，例如在Redis或者数据库中
 * 黑名单的生成往往有复杂的业务逻辑，具体情况算法不同
 * 但是在Spark Streaming进行处理的时候每次都能够访问完整的信息
 */
val blackList = Array(("Spy", true),("Cheater", true))
val blackListRDD = ssc.sparkContext.parallelize(blackList, 8)

val adsClickStream = ssc.socketTextStream("Master", 9999)

/**
 * 此处模拟的广告点击的每条数据的格式为：time、name
 * 此处map操作的结果是name、（time，name）的格式
 */
val adsClickStreamFormatted = adsClickStream.map { ads => (ads.split("")(1), ads) }
adsClickStreamFormatted.transform(userClickRDD => {
  // 通过leftOuterJoin操作既保留了左侧用户广告点击内容的RDD的所有内容，
  // 又获得了相应点击内容是否在黑名单中
val joinedBlackListRDD = userClickRDD.leftOuterJoin(blackListRDD)

  /**
   * 进行filter过滤的时候，其输入元素是一个元组：（name,((time,name), boolean)）
   * 其中，第一个元素是黑名单的名称
```

```
        * 第二元素的第二个元素boolean是进行leftOuterJoin的时候是否存在的值。
        * 如果存在，表明当前广告点击是黑名单，需要过滤掉，否则是有效点击内容
        */
       val validClicked = joinedBlackListRDD.filter(joinedItem => {
         if(joinedItem._2._2.getOrElse(false))
         {
           false
         } else {
           true
         }

       })

       validClicked.map(validClick => {validClick._2._1})
     }).print

     ssc.start()
     ssc.awaitTermination()

   }
}
```

此程序接收Socket信息，过滤掉其中名称为Spy、Cheater的信息，并打印输出。

把程序的批处理时间间隔设置从30s改成300s：

```
val ssc = new StreamingContext(conf, Seconds(300))
```

然后重新生成一下jar包。

Spark集群有5台机器：Master、Worker1、Worker2、Worker3、Worker4。

启动Spark的History Server。

打开数据发送的端口：

```
nc -lk 9999
```

用spark-submit运行前面生成的jar包。

在数据发送端口输入若干数据，例如：

```
1375864674543 Tom
1375864674553 Spy
1375864674571 Andy
1375864688436 Cheater
1375864784240 Kelvin
1375864853892 Steven
1375864979347 John
```

每行第一项为时刻的毫秒数，第二项是程序中要过滤的名称。

打开浏览器，看History Server的日志信息，如图1-2所示。

图中按时间顺序显示了曾经运行过的应用程序，第一列是App ID，有各应用程序执行信息的链接。

单击最新的应用，看目前运行的应用程序中有些什么Job，如图1-3所示。

这样一个Spark Streaming应用程序运行时总共有5个Job。

观察这些Job的内容，可以揭示一些现象。

Job 0不体现应用程序的业务逻辑代码，如图1-4所示。其实此Job是Spark Streaming出于对后面计算的负载均衡的考虑而产生的。

Spark 1.5.1 History Server

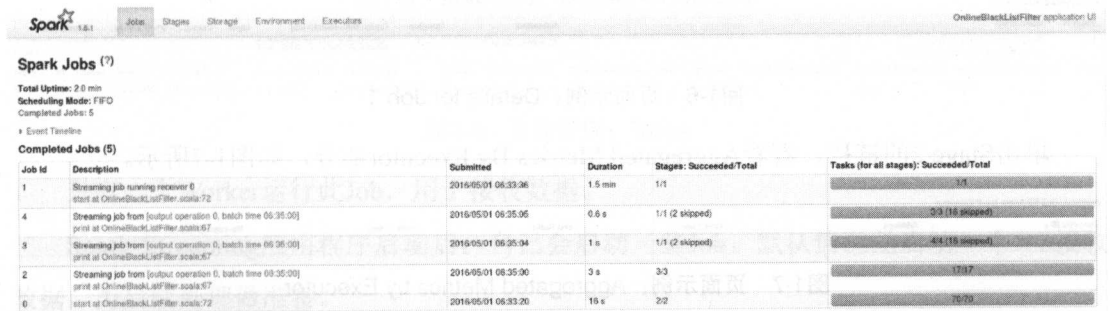
Event log directory: hdfs://Master:9000/historyserverforSpark

Showing 1-16 of 16

App ID	App Name	Started	Completed	Duration	Spark User	Last Up
app-20160430211417-0003	OnlineBlackListFilter	2016/04/30 21:14:13	2016/04/30 21:15:56	1.7 min	root	2016/04
app-20160430211216-0002	OnlineBlackListFilter	2016/04/30 21:12:12	2016/04/30 21:12:51	39 s	root	2016/04
app-20160430193900-0001	TransformBlacklist	2016/04/30 19:38:56	2016/04/30 19:44:32	5.6 min	root	2016/04
app-20160430193701-0000	TransformBlacklist	2016/04/30 19:36:53	2016/04/30 19:38:17	1.4 min	root	2016/04
local-1461736334126	NetworkWordCount	2016/04/27 13:52:06	2016/04/27 13:52:48	42 s	root	2016/04
local-1461736213566	NetworkWordCount	2016/04/27 13:50:05	2016/04/27 13:50:53	48 s	root	2016/04
local-1461736115514	NetworkWordCount	2016/04/27 13:48:24	2016/04/27 13:49:04	40 s	root	2016/04
app-20160427111949-0000	SparkSQL::192.168.112.235	2016/04/27 11:19:44	2016/04/27 13:47:06	2.5 h	root	2016/04
app-20160426140745-0000	Spark shell	2016/04/26 14:07:42	2016/04/26 14:09:03	1.3 min	root	2016/04
app-20160423133823-0003	MovieLensALS	2016/04/23 13:38:18	2016/04/23 13:51:24	13 min	root	2016/04
app-20160423133522-0002	MovieLensALS	2016/04/23 13:35:17	2016/04/23 13:35:40	23 s	root	2016/04
app-20160423132812-0001	MovieLensALS	2016/04/23 13:28:05	2016/04/23 13:28:15	10 s	root	2016/04
app-20160422132949-0000	Spark shell	2016/04/22 13:29:47	2016/04/22 13:30:34	47 s	root	2016/04
app-20160421072917-0001	FlumePushWordCount	2016/04/21 07:29:11	2016/04/21 07:29:54	43 s	root	2016/04
app-20160421072758-0000	FlumePushWordCount	2016/04/21 07:27:45	2016/04/21 07:29:05	1.3 min	root	2016/04
local-1461194198915	FlumePushWordCount	2016/04/21 07:16:19	2016/04/21 07:27:36	11 min	root	2016/04

Show incomplete applications

图1-2　History Server日志信息示例



Spark 1.5.1　Jobs　Stages　Storage　Environment　Executors　OnlineBlackListFilter application UI

Spark Jobs (?)

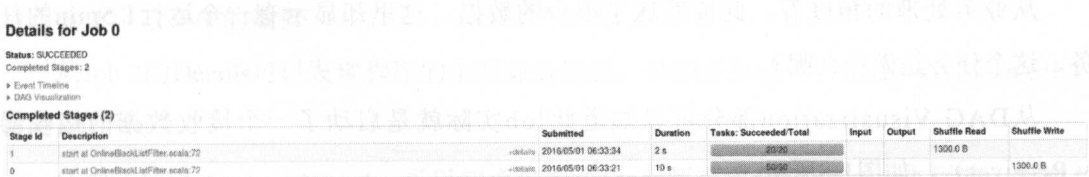
Total Uptime: 2.0 min
Scheduling Mode: FIFO
Completed Jobs: 5

▸ Event Timeline

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	Streaming job running receiver 0 start at OnlineBlackListFilter.scala:72	2016/05/01 06:33:36	1.5 min	1/1	1/1
4	Streaming job from [output operation 0, batch time 06:35:00] print at OnlineBlackListFilter.scala:67	2016/05/01 06:35:05	0.6 s	1/1 (2 skipped)	3/3 (16 skipped)
3	Streaming job from [output operation 0, batch time 06:35:00] print at OnlineBlackListFilter.scala:67	2016/05/01 06:35:04	1 s	1/1 (2 skipped)	4/4 (16 skipped)
2	Streaming job from [output operation 0, batch time 06:35:00] print at OnlineBlackListFilter.scala:67	2016/05/01 06:35:00	3 s	3/3	17/17
0	start at OnlineBlackListFilter.scala:72	2016/05/01 06:33:20	16 s	2/2	70/70

图1-3　Spark Jobs页面示例



Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

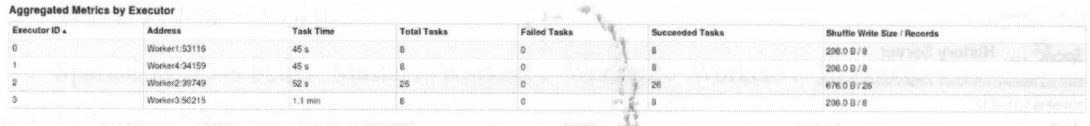
▸ Event Timeline
▸ DAG Visualization

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	start at OnlineBlackListFilter.scala:72 +details	2016/05/01 06:33:34	2 s	20/20			1300.0 B	
0	start at OnlineBlackListFilter.scala:72 +details	2016/05/01 06:33:21	10 s	50/50				1300.0 B

图1-4　Details for Job 0示例

Job 0包含Stage 0、Stage 1。随便看一个Stage，比如Stage 0，看看其中的Aggregated Metrics by Executor部分，如图1-5所示。



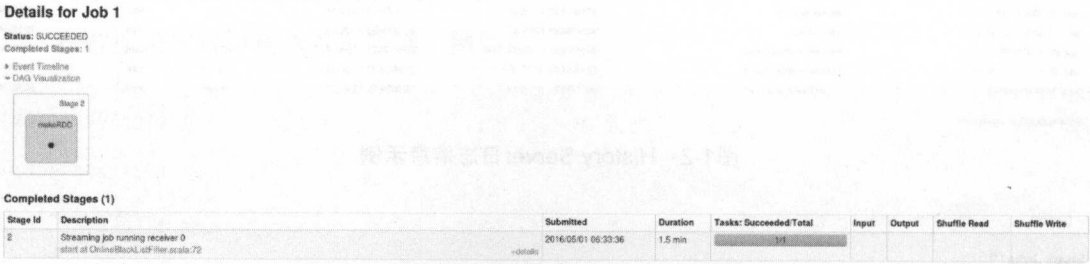
Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Write Size / Records
0	Worker1:53116	45 s	8	0	8	208.0 B / 8
1	Worker4:34159	45 s	8	0	8	208.0 B / 8
2	Worker2:39749	52 s	26	0	26	676.0 B / 26
3	Worker3:50215	1.1 min	8	0	8	208.0 B / 8

图1-5 Aggregated Metrics by Executor页面示例

因为是分布式环境做负载均衡，所以Job 0 的Stage 1是在4个Worker的Executor上运行。

Job 1的运行时间比较长，耗时1.5min，如图1-6所示。



Details for Job 1

Status: SUCCEEDED
Completed Stages: 1

▸ Event Timeline
▾ DAG Visualization

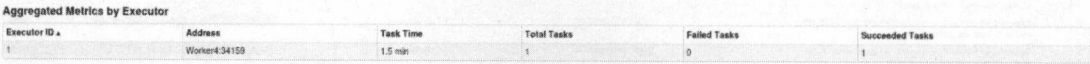
Stage 2
MapPartitionsRDD

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	Streaming job running receiver 0 start at OnlineBlackListFilter.scala:72	2016/05/01 06:33:36	1.5 min	1/1				

图1-6 页面示例：Details for Job 1

单击Stage 2的链接，看看Aggregated Metrics By Executor部分，如图1-7所示。



Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
1	Worker4:34159	1.5 min	1	0	1

图1-7 页面示例：Aggregated Metrics by Executor

可以知道，Stage 2只在Worker4上的一个Executor执行，而且执行了1.5min。

从业务处理的角度看，此前发送了很少的数据，这里却显示有一个运行1.5min的任务。这个任务是做什么呢？

从DAG Visualization部分可以知道此Job实际就是启动了一个接收数据的接收器（Receiver），如图1-8所示。

原来Receiver是通过一个Job来启动的。

Tasks部分如图1-9所示。

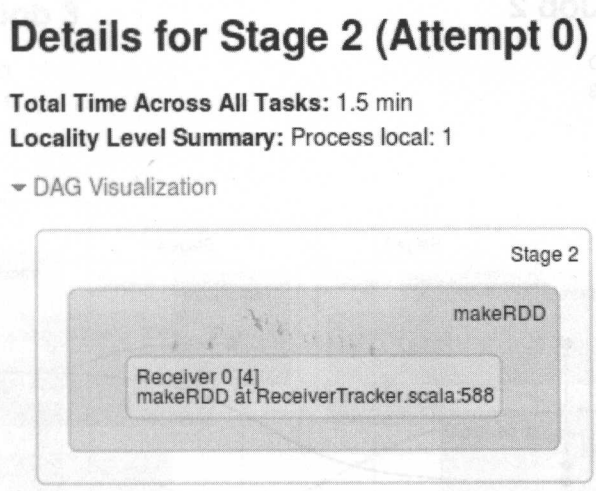


图1-8　页面示例：Details for Stage 2

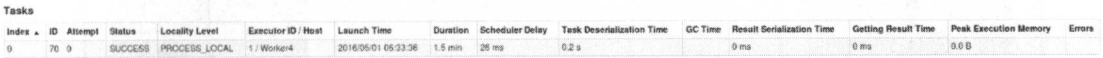


图1-9　页面示例：Tasks

只有一个Worker运行此Job，用于接收数据。

Spark Streaming应用程序启动后，自己会启动一些Job。默认情况是启动一个Job接收数据，为后续处理做准备。

Locality Level是PROCESS_LOCAL。所以，默认情况下，数据接收不会使用磁盘，而是直接使用内存中的数据。

从Job 2的Details可以发现程序的主要业务逻辑，体现在Stage 3、Stage 4、Stage 5中，如图1-10所示。

仔细观察Stage 3、Stage 4，可以知道这两个Stage都是用4个Executor执行的，所有数据处理是在4台机器上进行的，如图1-11所示。

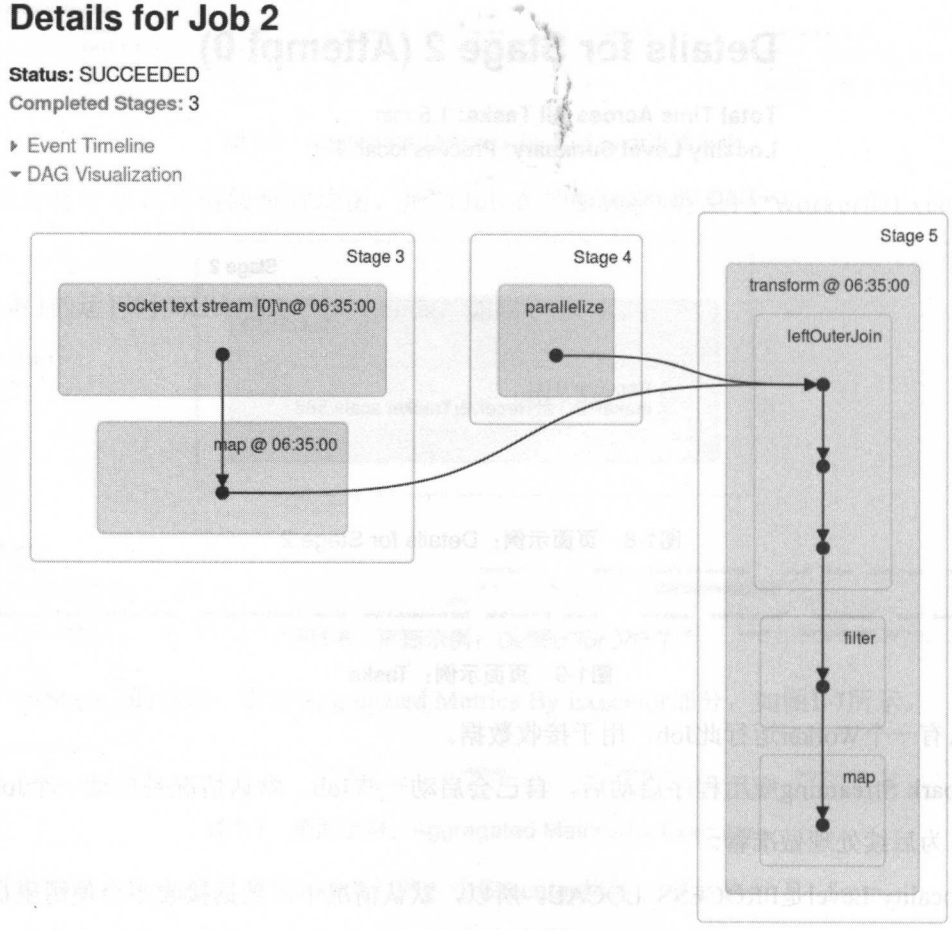


图1-10　页面示例：Details for Job 2

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Write Size / Records
0	Worker1:53116	2 s	1	0	1	152.0 B / 1
1	Worker4:34159	8 s	3	0	3	466.0 B / 3
2	Worker2:39749	4 s	2	0	2	309.0 B / 2
3	Worker3:50215	6 s	2	0	2	314.0 B / 2

图1-11　页面示例：Aggregated Metrics by Executor

Stage 5只在Worker4上，这是因为这个Stage有Shuffle操作。

Job 3有Stage 6、Stage 7、Stage 8，其中Stage 6、Stage 7显示灰色，说明被跳过，如图1-12所示。

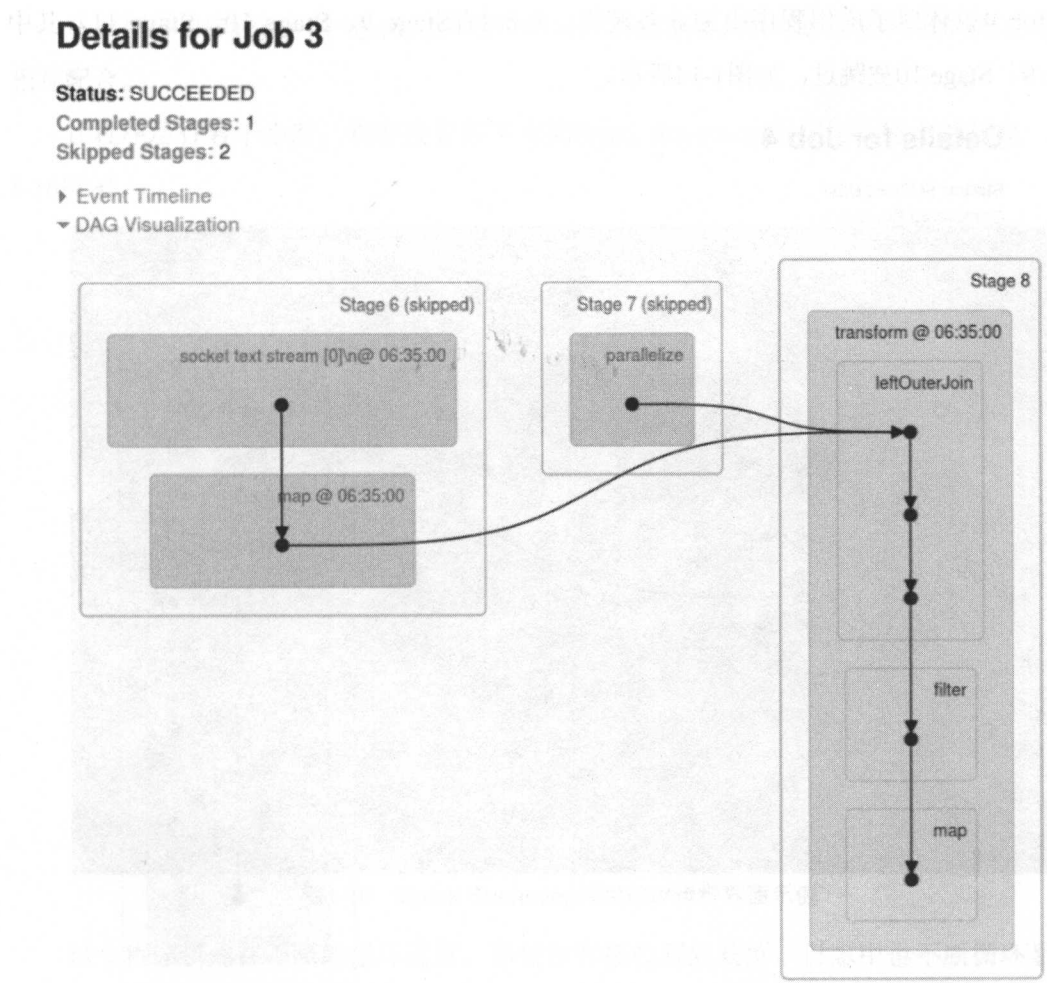


图1-12　页面示例：Details for Job 3

由Stage 8的Aggregated Metrics by Executor部分可以看出，数据处理是在4台机器上进行的，如图1-13所示。

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
0	Worker1:53116	1 s	1	0	1	160.0 B / 0
1	Worker4:34159	0.9 s	1	0	1	184.0 B / 1
2	Worker2:39749	0.8 s	1	0	1	176.0 B / 1
3	Worker3:50215	1 s	1	0	1	160.0 B / 0

图1-13　页面示例：Aggregated Metrics by Executor

Job 4也体现了应用程序中的业务逻辑。Job 4有Stage 9、Stage 10、Stage 11，其中Stage 9、Stage 10被跳过，如图1-14所示。

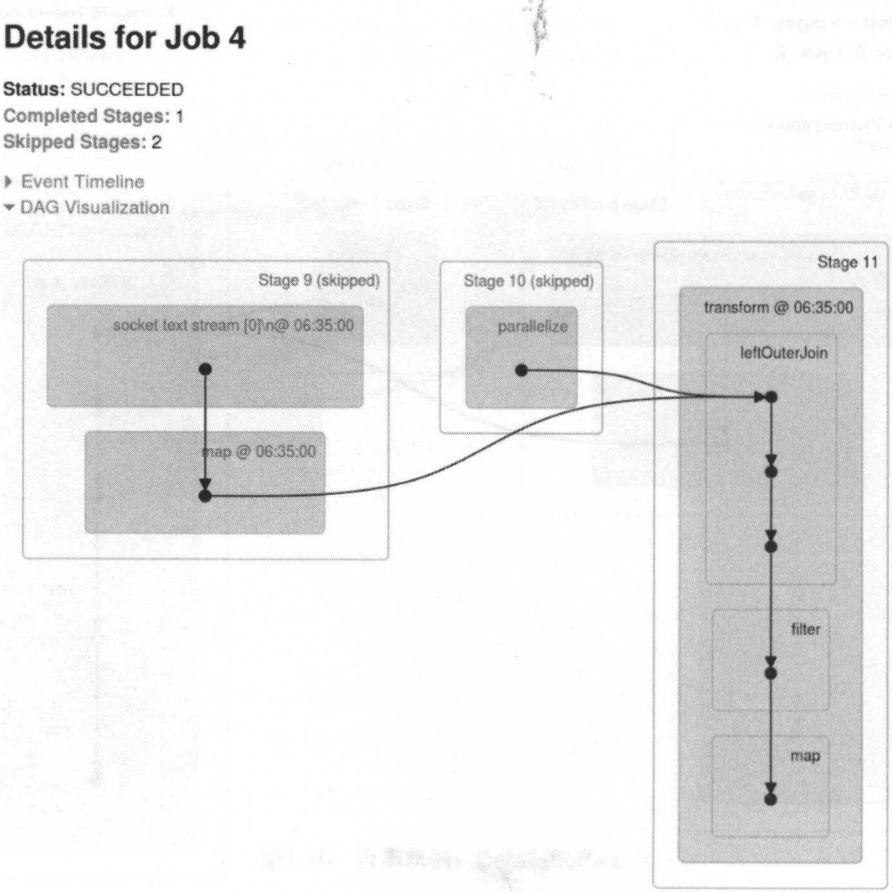


图1-14　页面示例：Details for Job 4

细察Stage 11，可以看出，数据处理是在Worker2之外的其他3台机器上进行的，如图1-15所示。

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
0	Worker1:53116	0.4 s	1	0	1	160.0 B / 0
1	Worker4:34159	0.4 s	1	0	1	235.0 B / 3
3	Worker3:50215	0.6 s	1	0	1	215.0 B / 2

图1-15　页面示例：Aggregated Metrics by Executor

从而可得出结论：一个Spark应用程序中可以启动很多Job，而这些不同的Job之间可以相互配合。

让程序运行若干小时，观察没有停下来的Spark Streaming程序运行留下的信息，如图1-16所示。

图1-16　Spark Streaming应用程序运行界面示例

这个程序仍然在不断地循环运行。即使没有接收到新数据，日志中也不断循环显示着JobScheduler、BlockManager、MapPartitionsRDD、ShuffledRDD等的信息，其中有一部分是Spark Core相关的信息。

1.2　Spark Streaming应用剖析

Spark Streaming可以接收Kafka、Flume、HDFS和Kinesis等各种来源的实时输入数

据，进行处理后，处理结果保存在HDFS、Databases等各种地方，如图1-17所示。

图1-17　Spark Streaming的数据流

Spark Streaming接收这些实时输入数据流，会将它们按批次划分，然后交给Spark引擎处理，生成按照批次划分的结果流，如图1-18所示。

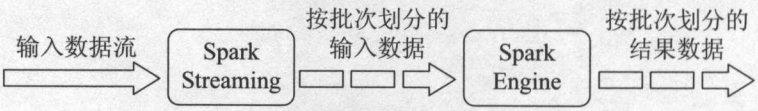
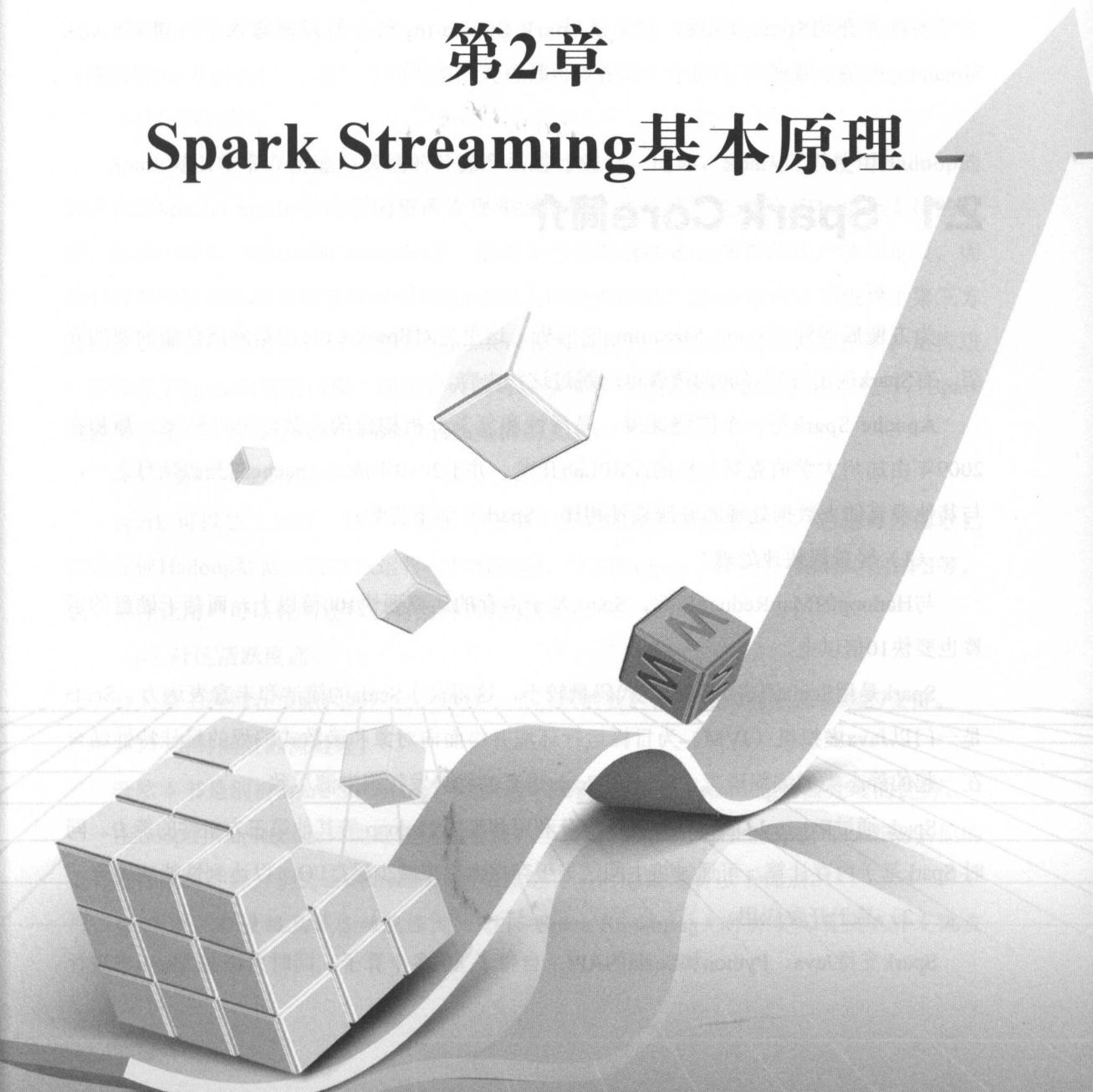


图1-18　Spark Streaming在Spark内的数据流

Spark Streaming程序代码中使用了StreamingContext作为Spark Streaming的入口，还使用了高度抽象的map、flatMap、reduce、join等原语，对初始的DStream（discretized stream，离散流）进行转换生成新的DStream，并利用最后的DStream来保存处理结果。

Spark Streaming为接收数据、负载均衡、数据处理会分别产生并执行若干个Job。

第2章
Spark Streaming基本原理



第1章给出了一个Spark Streaming应用程序的运行过程，并显示了若干日志信息。即使读者没写过类似的代码，结合注释去阅读，也容易理解。至于产生那些Job的原因，通过后续章节的学习，都会有答案。但是读者只有了解了Spark和RDD的基本原理，才能顺利地开始后续的阅读。

本章先介绍Spark Core，然后从Spark Streaming引入的新概念入手，讲解Spark Streaming的基本原理。

2.1 Spark Core简介

为方便后面针对Spark Streaming的剖析，这里先对Spark Core的基础概念做简要的介绍，有Spark应用开发经验的读者可以跳过这些内容。

Apache Spark是一个围绕速度、易用性和复杂分析构建的大数据处理框架，最初在2009年由加州大学伯克利分校的AMPLab开发，并于2010年成为Apache的开源项目之一，与其他现有的大数据处理的开源软件相比，Spark有如下优势：

（1）轻量级快速处理。

与Hadoop的MapReduce相比，Spark基于内存的运算要快100倍以上，而基于硬盘的运算也要快10倍以上。

Spark是用Scala语言编写的，代码量较小，这得益于Scala的简洁和丰富表达力。Scala是一门以Java虚拟机（JVM）为目标运行环境并将面向对象和函数式编程的最佳特性结合在一起的静态类型编程语言。Spark把Scala语言的优势发挥得淋漓尽致。

Spark 通过External DataSource API 充分利用和集成 Hadoop 等其他第三方组件的能力。同时 Spark基于内存计算，可通过将中间结果缓存在内存来减少磁盘I/O，以达到性能的提升。

（2）易于开发应用。

Spark支持Java、Python和Scala的API，自带了 80 多个算子，同时允许在 Shell 中进行

交互式计算。用户可以利用 Spark 像书写单机程序一样书写分布式程序，轻松利用 Spark 搭建大数据内存计算平台并充分利用内存计算实现海量数据的实时处理。

（3）高通用性。

Spark提供了统一的解决方案，Spark可以用于批处理、交互式查询（通过Spark SQL）、实时流处理（通过Spark Streaming）、机器学习（通过Spark MLlib）和图计算（通过Spark GraphX）。这些不同类型的处理都可以在同一个应用中无缝使用。

（4）可融合性。

Spark可以非常方便地与其他的开源产品进行融合。比如，Spark可以使用Hadoop的YARN和Apache Mesos作为它的资源管理和调度器，并且可以处理所有Hadoop支持的数据，包括HDFS、HBase和Cassandra等。这对于已经部署Hadoop集群的用户特别重要，因为不需要做任何数据迁移就可以使用Spark强大的处理能力。Spark也可以不依赖于第三方的资源管理器和调度器，它实现了Standalone作为其内置的资源管理和调度框架，这样进一步降低了Spark的使用门槛，使所有人都可以非常容易地部署和使用Spark。此外，Spark还提供了在EC2上部署Standalone的Spark集群的工具。

（5）支持多数据源。

Spark 可以独立运行，除了可以运行在当下的YARN集群管理之外，它还可以读取已有的任何Hadoop数据。它可以运行多种数据源，比如Parquet、Hive、HBase、HDFS等。这个特性让用户可以轻易迁移已有的持久化层数据。

（6）社区活跃度高。

有大量工程师在贡献代码，已经形成一个活跃的社区，有强大的生态系统的支持。

（7）实时高效的流处理。

既然本书是剖析Spark Streaming，就应该单独说说Spark Streaming的流处理的优势。

Hadoop的MapReduce只能处理离线数据，当然在YARN之后Hadoop也可以借助其他的工具进行流式计算。但Spark Streaming对数据进行实时的处理有以下优点：

- 简单。轻量级且具备功能强大的API，Spark Streaming允许开发人员快速开发流应用程序。

- 集成。支持多种来源的流数据。为流处理和批处理重用了同样的代码，甚至可以将流数据保存到历史数据中。能调用Spark SQL、MLlib、GraphX等其他子框架来实现多种数据处理功能。
- 容错。相比其他的流解决方案，Spark Streaming无须额外的代码和配置，就可以做大量的恢复和交付工作。

Spark Core中最重要的概念是RDD（弹性分布式数据集）。

RDD是由数据组成的不可变分布式集合，有以下组成部分：

- 一组分区（partition）。
- 一个函数，用于做每个分区上的计算。
- 一个对其他RDD的依赖关系列表。
- 一个对key-value（键值对）类型的RDD的分区器（可选）。
- 一个所有分区的优选位置的列表（可选）。

RDD可以快速和便捷地转换到集群中的主机上。这也就是RDD的弹性所在。RDD虽然不可变，但Spark能对RDD进行转换操作生成新的RDD。RDD是一个可分区的元素集合，其包含的元素可以分布在集群各个节点上，并且可以执行一些分布式并行操作。

这里给出Spark架构相关的术语及其解释：

- Application（应用程序）。Spark Application的概念和Hadoop MapReduce中的类似，指的是用户编写的Spark应用程序，包含了一个Driver 功能的代码和分布在集群中多个节点上运行的Executor代码。
- Driver（驱动器）。Spark Application的main函数在运行时会创建SparkContext。通常用SparkContext代表Driver。其中创建SparkContext的目的是为了准备Spark应用程序的运行环境。在Spark中由SparkContext负责和ClusterManager通信，进行资源的申请、任务的分配和监控等；当Executor部分运行完毕后，Driver负责将SparkContext关闭。
- Executor（执行器）。Application运行在Worker 节点上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上。每个Application

都有各自独立的一批Executor。在Spark on Yarn模式下，其进程名称为CoarseGrainedExecutorBackend，类似于Hadoop MapReduce中的YarnChild。一个CoarseGrainedExecutorBackend进程有且仅有一个executor对象，它负责将Task包装成taskRunner，并从线程池中抽取出一个空闲线程运行Task。每个CoarseGrainedExecutorBackend能并行运行Task的数量取决于分配给它的CPU的个数。

- ClusterManager（集群管理器）。指的是在集群上获取资源的外部服务，目前有以下两种：
  - Standalone。Spark原生的资源管理，由Master负责资源的分配。
  - Hadoop Yarn。由YARN中的ResourceManager负责资源的分配。
- Worker（工作者）。集群中任何可以运行Application代码的节点，类似于YARN中的NodeManager节点。在Standalone模式中指的就是通过Slave文件配置的Worker节点，在Spark on Yarn模式中指的就是NodeManager节点。
- Job（作业）。包含多个Task组成的并行计算，往往由Spark action催生，一个Job包含多个RDD及作用于相应RDD上的各种操作。
- Stage（阶段）。每个Job会被拆分为很多组Task，每组Task被称为Stage，也可称TaskSet，一个Job分为多个阶段。
- Task（任务）。被送到某个Executor上的工作任务。
- DAG（Directed Acyclic Graph，有向无环图）。RDD及其依赖关系构成的图。
- DAGScheduler（DAG调度器）。根据Job构建基于Stage的DAG，并提交Stage给TaskScheduler。
- TaskScheduler（任务调度器）。将Taskset提交给Worker节点集群运行并返回结果。

可以通过图2-1所示的Spark架构加深对上面主要术语的理解。

总体来说，每个Spark应用都包含一个Driver，Driver运行用户的main函数，并在集群上执行各种并行操作。

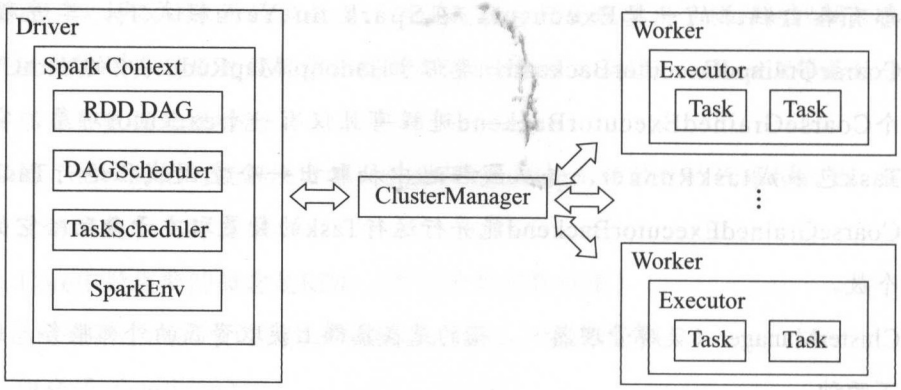


图2-1 Spark架构

这里给出简单代码示例。

源码2-1 Spark应用程序WordCount

```
package com.example


import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._


object WordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: <file>")
      System.exit(1)
    }


    val conf = new SparkConf()
```

```
    val sc = new SparkContext(conf)
    val line = sc.textFile(args(0))

    line.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_).collect().foreach(println)

    sc.stop()
  }
}
```

RDD通常是通过HDFS（或者其他Hadoop支持的文件系统）上的文件或者Driver中的Scala集合对象来创建或转换（transformation）得到的。用户也可以请求Spark将RDD持久化到内存里，以便在不同的并行操作里被复用。RDD具备容错性，可以从节点失败中自动恢复数据。

源码2-1的WordCount通过sc.textFile产生了第一个RDD，然后通过其他操作转换成其他RDD，最后一定有动作（action）操作。本例中action操作是collect、foreach。

RDD作为数据结构，本质上是一个只读的分区记录集合。一个RDD可以包含多个分区（partition），每个分区就是一个dataset片段。

Spark中对RDD的操作分为两类：转换（transformation）和动作（action）。

RDD的transformation操作如表2-1所示。

表2-1 RDD的transformation操作

转 换	描 述
map(func)	返回一个新的分布式数据集，其中每个元素都是由源RDD中一个元素经func转换得到的
filter(func)	返回一个新的数据集，其中包含的元素来自源RDD中元素经func过滤后（func返回true时才选中）的结果
flatMap(func)	类似于map，但每个输入元素可以映射到0～*n*个输出元素（所以要求func必须返回一个Seq而不是单个元素）

续表

转　换	描　述
mapPartitions(func)	类似于map，但基于每个RDD分区（或者数据block）独立运行，所以如果RDD包含元素类型为T，则 func 必须是 Iterator<T> => Iterator<U> 的映射函数
mapPartitionsWithIndex(func)	类似于 mapPartitions，只是func 多了一个整型的分区索引值，因此如果RDD包含元素类型为T，则 func 必须是 Iterator<T> => Iterator<U> 的映射函数
sample(withReplacement, fraction, seed)	采样部分（比例取决于 fraction）数据，同时可以指定是否使用回置采样（withReplacement）以及随机数种子（seed）
union(otherDataset)	返回源数据集和参数数据集（otherDataset）的并集
intersection(otherDataset)	返回源数据集和参数数据集（otherDataset）的交集
distinct([numTasks]))	返回对源数据集做元素去重后的新数据集
groupByKey([numTasks])	只对包含键值对的RDD有效，如源RDD包含（K，V） 对，则该算子返回一个新的数据集包含（K， Iterable<V>）对。 注意：如果需要按key分组聚合（如sum或average），推荐使用reduceByKey或aggregateByKey 以获得更好的性能。 注意：默认情况下，输出计算的并行度取决于源RDD的分区个数。当然，也可以通过设置可选参数 numTasks 来指定并行任务的个数
reduceByKey(func, [numTasks])	如果源RDD包含元素类型（K，V） 对，则该算子也返回包含（K，V）对的RDD，只不过每个key对应的value是经过func聚合后的结果，而func本身是一个（V，V） => V 的映射函数。 另外，和 groupByKey 类似，可以通过可选参数 numTasks 指定reduce任务的个数
aggregateByKey(zeroValue) (seqOp, combOp, [numTasks])	如果源RDD包含（K，V） 对，则返回新RDD包含（K，U）对，其中每个key对应的value都是由 combOp 函数和一个0值（zeroValue）聚合得到。允许聚合后value类型和输入value类型不同，避免了不必要的开销。和 groupByKey 类似，可以通过可选参数 numTasks 指定reduce任务的个数
sortByKey([ascending], [numTasks])	如果源RDD包含元素类型（K，V） 对，其中K可排序，则返回新的RDD包含（K，V）对，并按照 K 排序（升序还是降序取决于 ascending 参数）
join(otherDataset, [numTasks])	如果源RDD包含元素类型（K，V）且参数RDD（otherDataset）包含元素类型（K，W），则返回的新RDD中将包含内关联后key对应的（K，（V，W））对。外关联（outer join）操作请参考 leftOuterJoin、rightOuterJoin以及fullOuterJoin算子
cogroup(otherDataset, [numTasks])	如果源RDD包含元素类型（K，V）且参数RDD（otherDataset）包含元素类型（K，W），则返回的新RDD中包含（K，(Iterable<V>，Iterable<W>)）。该算子还有个别名：groupWith
cartesian(otherDataset)	如果源RDD包含元素类型 T 且参数RDD（otherDataset）包含元素类型 U，则返回的新RDD包含前二者的笛卡儿积，其元素类型为（T，U）对
pipe(command, [envVars])	以shell命令行管道处理RDD的每个分区，如Perl 或者 bash 脚本。RDD中每个元素都将依次写入进程的标准输入（stdin），然后按行输出到标准输出（stdout），每一行输出字符串即成为一个新的RDD元素
coalesce(numPartitions)	将RDD的分区数减少到numPartitions。当以后大数据集被过滤成小数据集后，减少分区数，可以提升效率

续表

转 换	描 述
repartition(numPartitions)	将RDD数据重新混洗（reshuffle）并随机分布到新的分区中，使数据分布更均衡，新的分区个数取决于numPartitions。该算子总是需要通过网络混洗所有数据
repartitionAndSortWithin-Partitions(partitioner)	根据partitioner（Spark自带HashPartitioner和RangePartitioner等）重新分区RDD，并且在每个结果分区中按key做排序。这是一个组合算子，功能上等价于先 repartition 再在每个分区内排序，但这个算子内部做了优化（将排序过程下推到混洗同时进行），因此性能更好

RDD的这些转换操作是从父RDD转换为子RDD。注意，这里说的RDD的父子关系不是指RDD类的继承关系，而是指RDD对象的转换关系。Spark Streaming内部为了优化最终的分布式运算，把子RDD对父RDD的这种依赖分成宽依赖和窄依赖两类。

- 窄依赖（narrow dependency）：父RDD的每个分区都只被子 RDD 的一个分区所依赖。
- 宽依赖（wide dependency）：父RDD的每个分区被多个子 RDD 的分区所依赖。

RDD的转换操作中的groupByKey、reduceByKey、join等会导致宽依赖。

RDD的action操作如表2-2所示。

表2-2 RDD的action操作

动 作	描 述
reduce(func)	将RDD中的元素按func进行聚合（func是一个（T，T）=> T 的映射函数，其中T为源RDD元素类型，并且func需要满足交换律和结合律以便支持并行计算）
collect()	将数据集中所有元素以数组形式返回驱动器（driver）程序。通常用于在RDD进行了filter或其他过滤操作后，将一个足够小的数据子集返回到驱动器内存中
count()	返回数据集中元素个数
first()	返回数据集中首个元素（类似于 take(1)）
take(n)	返回数据集中前 n 个元素
takeSample(withReplacement, num, [seed])	返回数据集的随机采样子集，最多包含 num 个元素，withReplacement 表示是否使用回置采样，最后一个参数为可选参数seed，是随机数生成器的种子
takeOrdered(n, [ordering])	按元素排序（可以通过 ordering 自定义排序规则）后，返回前 n 个元素
saveAsTextFile(path)	将数据集中的元素保存到指定目录下的文本文件（或者多个文本文件）中，支持本地文件系统、HDFS 或者其他任何Hadoop支持的文件系统。保存过程中，Spark会调用每个元素的toString方法，并将结果保存成文件中的一行

续表

动　作	描　述
saveAsSequenceFile(path) (Java and Scala)	将数据集中的元素保存到指定目录下的Hadoop Sequence文件中，支持本地文件系统、HDFS 或者其他任何Hadoop支持的文件系统。适用于实现了Writable接口的键值对RDD。在Scala中，同样也适用于能够被隐式转换为Writable的类型（Spark实现了所有基本类型的隐式转换，如Int、Double、String 等）
saveAsObjectFile(path) (Java and Scala)	将RDD元素以Java序列化的格式保存成文件，保存结果文件可以使用SparkContext.objectFile来读取
countByKey()	只适用于包含键值对（K，V）的RDD，并返回一个哈希表，包含（K，Int）对，表示每个key的个数
foreach(func)	在RDD的每个元素上运行 func 函数。通常被用于累加操作，如更新一个累加器（Accumulator）或者和外部存储系统互操作 注意：用 foreach 操作除累加器之外的变量可能导致未定义的行为

转换操作不会立即执行，只有在action操作时，通过SparkContext执行提交作业的runJob操作，触发RDD DAG的执行。

RDD还有persist()、cache()等操作，持久化（或缓存）数据集到内存或磁盘。被缓存的 RDD在被使用的时候，存取速度会被大大加速。RDD的persist()方法可以使用多种缓存策略。RDD的cache()方法其实调用的就是persist()方法，缓存策略为MEMORY_ONLY。

图2-2展示了一个Spark应用案例的计算方式。

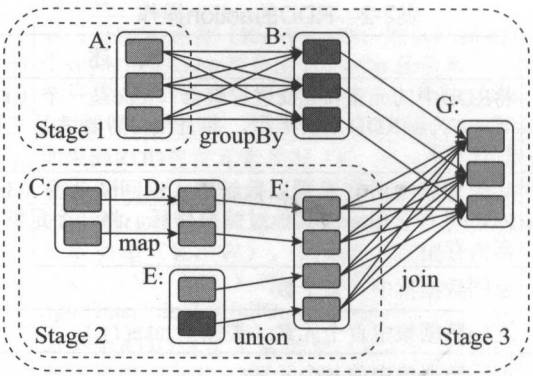


图2-2　Spark应用案例的计算方式

宽依赖是划分出Stage的依据。图2-2所示的案例由于有groupBy、join导致宽依赖，所以划分出了3个Stage。Stage有非最终的Stage（Shuffle Map Stage）和最终的Stage（Result Stage）两种，Stage的边界就是发生shuffle的地方。

计算方式确定下来后，会根据Stage生成TaskSet，TaskSet被提交并执行。

Spark第二个重要的抽象概念是共享变量，共享变量是一种可以在并行操作之间共享使用的变量。默认情况下，当Spark把一系列任务调度到不同节点上运行时，Spark会同时把每个变量的副本和任务代码一起发送给各个节点。但有时候需要在任务之间或者任务和Driver之间共享一些变量。

Spark提供了两种类型的共享变量：

（1）广播变量。在所有节点的内存中缓存一个值。广播变量是只读变量。

代码示例：

```
val broadcastVar = sc.broadcast("string for test")
val v = broadcastVar.value
println(v)
```

（2）累加器。用来执行跨节点的“累加”操作，例如计数和求和。

代码示例：

```
val accum = sc.accumulator(0, "My Accumulator")
sc.parallelize(1 to 1000000).foreach(x => accum+= 1)
println(accum.name + " : " + accum.value)
```

最后总结一下Spark应用程序的运行流程：

（1）构建Spark Application的运行环境，启动SparkContext。

（2）SparkContext向资源管理器（可以是Standalone、Mesos、Yarn）申请运行Executor资源。

（3）资源管理器分配Executor资源并启动ExecutorBackend，Executor运行情况将随着心跳发送到资源管理器上。心跳是周期性地发送给资源管理器的信息，以表示Executor仍然活着。

（4）SparkContext通过DAGScheduler根据RDD依赖关系构建DAG图，再将DAG图分解成Stage，并把Taskset发送给TaskScheduler。Executor向SparkContext申请Task，TaskScheduler将Task发放给Executor运行。

（5）Task在Executor上运行，运行完释放所有资源。

以上对Spark Core作了概括性的介绍。没有掌握这些基本知识的读者还可以参考其他资料。

2.2 Spark Streaming设计思想

第1章对Spark Streaming应用的外部表象做了介绍。那么其内部是怎么运作的呢？

Spark Streaming提供了表示连续数据流的、高度抽象的离散流（DStream）。DStream本质上是对RDD的一层封装。DStream中的每个RDD都包含来自一个时间间隔的数据，如图2-3所示。

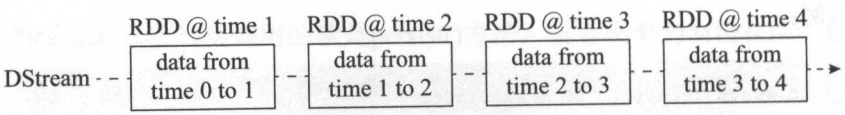


图2-3 DStream与RDD的关系

DStream是一个没有边界的集合，没有大小的限制。

DStream代表了时空的概念。图2-3就体现了多个时间段。具体到每个时间段，就是空间的操作，也就是对时间间隔的对应批次的数据的处理。

Spark Streaming应用程序中，除了使用数据源产生的数据流来创建DStream，也会在已有的DStream上使用某种操作来创建新的DStream。图2-4显示的是对行DStream做了flatMap操作，生成单词DStream。

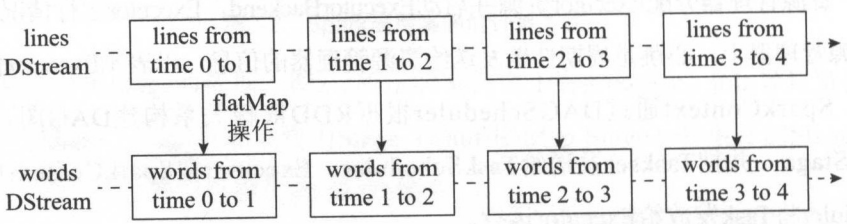
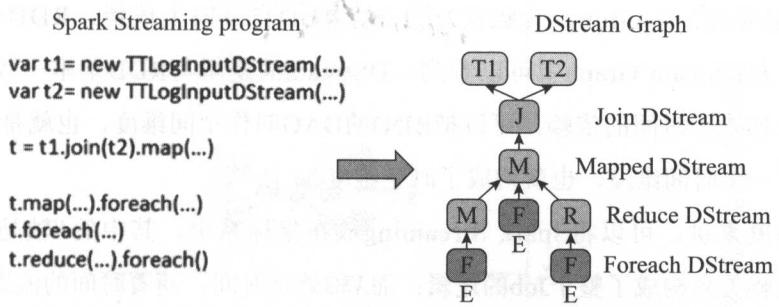


图2-4 利用flatMap操作把行DStream转换为单词DStream的示例

下面用实例来讲解数据处理过程。图2-5（a）显示了实例代码的主要部分。

Spark Streaming程序中一般会有若干个对DStream的操作。DStreamGraph就是由这些操作的依赖关系构成的。Spark Streaming程序转换为Spark执行的作业的过程中使用了DStreamGraph。

从程序到DStreamGraph的转换如图2-5所示。图2-5（a）所示的代码和图2-5（b）所示的DStream Graph是相对应的。



（a）代码的主要部分　　（b）相应的DStream Graph

图2-5　案例：Spark Streaming代码与DStream Graph的对应关系

本例代码首先用来生成DStream Graph。前面的操作不会马上执行，而是从每个foreach开始都会进行回溯。从后往前回溯这些操作之间的依赖关系，也就形成了DStream Graph。

执行从DStream到RDD的转换也就形成了RDD Graph，如图2-6所示。

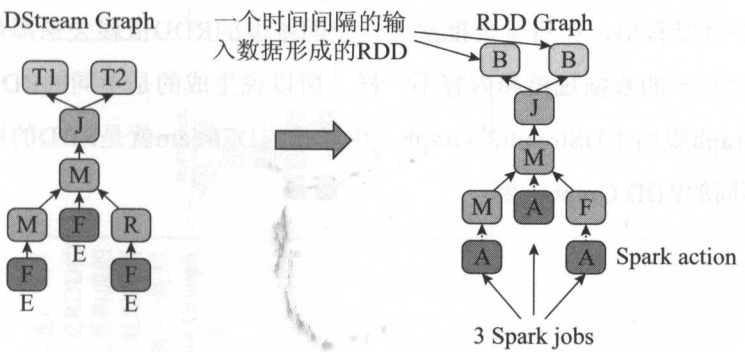


图2-6　案例：执行从DStream到RDD的转换就形成了RDD Graph

空间维度确定之后，随着时间的不断推进，会不断实例化RDD Graph，然后触发Job去执行处理。

Spark Core离不开RDD，Spark Streaming自然也离不开。RDD之间的具体依赖构成了空间维度。DStream就是在RDD的基础上增加了时间维度。所以整个Spark Streaming就是时空维度的。

Spark Core处理的每一步都是基于RDD的，RDD之间有依赖关系。图2-6的例子中，RDD的DAG显示有3个action，会触发3个Job，RDD自下向上依赖，RDD产生Job就会具体地执行。从DSteam Graph中可以看到，DStream的逻辑与RDD基本一致，它就是在RDD的基础上加上了时间的依赖。可以把RDD的DAG叫作空间维度，也就是说整个Spark Streaming多了一个时间维度，也就构成了时空维度。

从这个角度来讲，可以将Spark Streaming放在坐标系中。其中的*Y*轴是对RDD的操作，RDD的依赖关系构成了整个Job的逻辑；而*X*轴就是时间，随着时间的流逝，固定的批处理时间间隔（Batch Interval）就会生成一个Job实例，进而在集群中运行。

图2-7是批处理周期为1s的Spark Streaming应用程序在时空维度中的运行图示。

对于Spark Streaming来说，当不同的数据来源的数据流进来的时候，基于固定的时间间隔，会形成一系列固定不变的数据集或event集合（例如来自flume和kafka）。而这正好与RDD基于固定的数据集不谋而合，事实上，由DStream基于固定的时间间隔形成的RDD Graph正是基于某一个批次的数据集的。

从图2-7中可以看出，在每一个批次上，空间维度的RDD依赖关系都是一样的，不同的是这5个批次流入的数据规模和内容不一样，所以说生成的是不同的RDD依赖关系的实例。RDD的Graph脱胎于DStream的Graph，也就是说DStream就是RDD的模板，不同的时间间隔生成不同的RDD Graph实例。

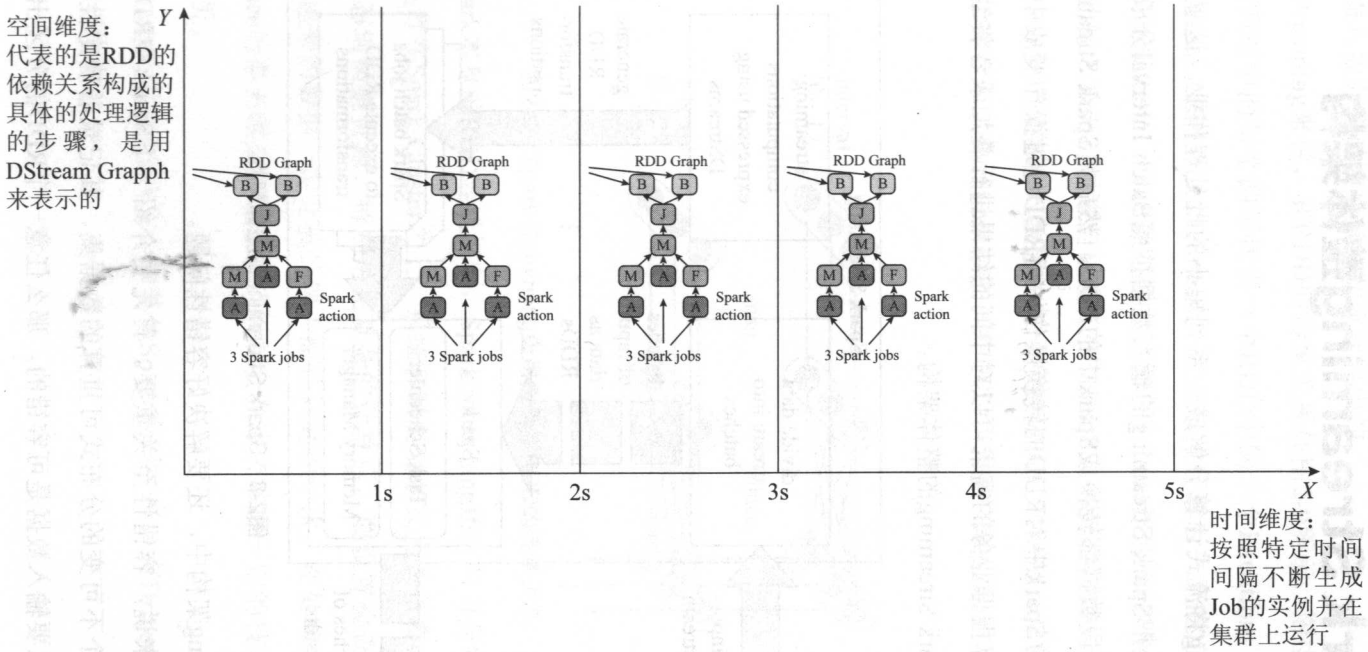


图2-7　Spark Streaming应用程序不断产生Job并执行

2.3 Spark Streaming整体架构

Spark Streaming将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark Core，也就是把Spark Streaming的输入数据按照Batch Interval分成一段一段的数据（DStream），每一段数据都转换成Spark中的RDD，然后将Spark Streaming中对DStream的转换操作变为针对Spark中对RDD的转换操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者将结果存储到外部设备。图2-8显示了Spark Streaming的整体架构。

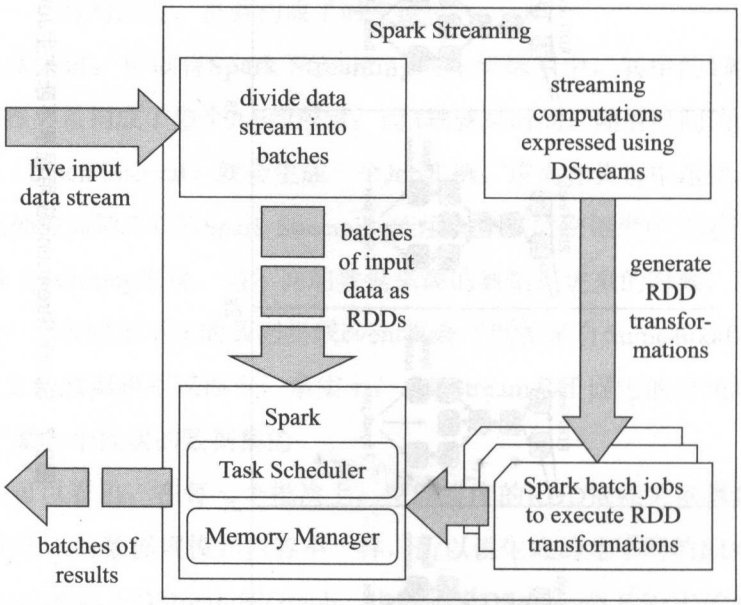


图2-8 Spark Streaming的整体架构

在Spark Streaming架构中，还要解决好容错性问题。

对于流式计算来说，容错性至关重要。首先要介绍一下Spark中RDD的容错机制。每一个RDD都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（lineage），所以只要输入数据是可容错的，那么任意一个RDD的分区出错或不可用，都

是可以利用原始输入数据通过转换操作而重新算出的。

对于Spark Streaming来说，其RDD的传承关系如图2-9所示，图中的每一个椭圆形表示一个RDD，椭圆形中的每个圆形代表一个RDD中的一个Partition，图中每一列的多个RDD表示一个DStream（图中有3个DStream），而每一行最后一个RDD则表示每一个Batch Size所产生的中间结果RDD。可以看到图中的每一个RDD都是通过lineage相连接的，由于Spark Streaming的输入数据可以来自磁盘，例如HDFS（多份复制）或是来自网络的数据流（Spark Streaming会将网络输入数据的每一个数据流复制两份到其他的机器）都能保证容错性，所以RDD中任意的Partition出错，都可以并行地在其他机器上将缺失的Partition计算出来。

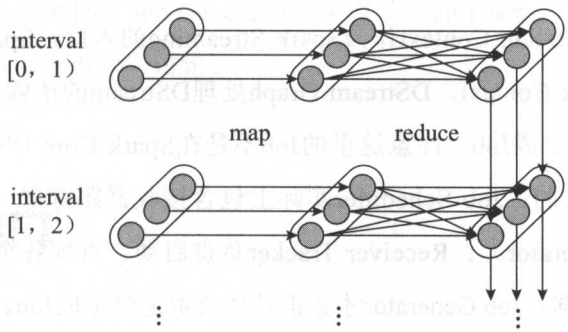


图2-9 Spark Streaming应用案例中RDD的传承关系

除了以上Spark本身的RDD容错机制，Spark Streaming还有与自身特点相关的容错问题需要解决。Job运行在Spark Cluster之上，系统容错更复杂又至关重要。计算性能不好时，必须能限流和动态地调整资源。特别是在复杂的计算后，要设置检查点（checkpoint）。有时我们希望流进来的数据一定会被处理，而且只处理一次。在处理出现崩溃的情况下，要保证exactly-once的事务语义。这些容错机制会在读者掌握一定的技术后专门在后面的章节中进行详细分析。

Spark Streaming运行在Spark上，和其他Spark应用一样，有Driver、Worker、Executor等部分，如图2-10所示。

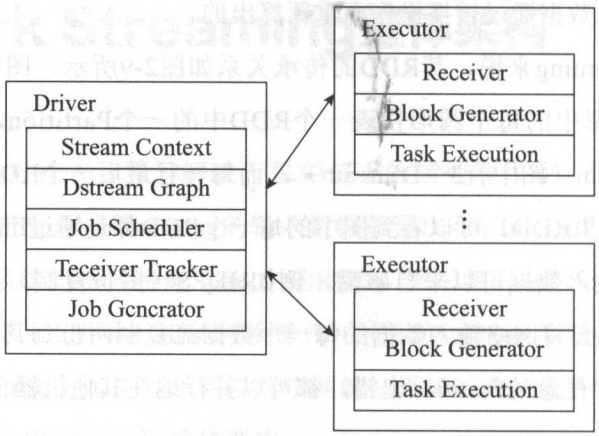


图2-10 Spark Streaming应用架构中的Driver和Executor

在Driver中，有Stream Context作为Spark Streaming的入口，Spark Streaming的最终处理实际还是交给Spark Context。DStream Graph处理DStream的依赖关系。Job调度器（Job Scheduler）负责定时生成Job。注意这里的Job不是在Spark Core 中提到的Job，而是Spark Streaming中自己定义的。Job Scheduler实际上包含接收器跟踪器（Receiver Tracker）和Job生成器（Job Generator）。Receiver Tracker负责启动、管理各个 Receiver 及管理各个 Receiver 接收到的数据。Job Generator才真正具体负责定时生成Job。

在Worker中一般会生成若干个Executor。接收器（Receiver）运行在Executor中，接收的数据通过块生成器（Block Generator）生成块（Block）。在每个批处理时间间隔（Batch Duration）中都有从Driver提交来的Job的业务逻辑在Executor中被执行。

无论是Block数据的生成还是Job的生成，都需要能无休止地周而复始地进行下去。Spark Streaming中设计了一个RecurringTimer类来起到定时器的作用。Executor上的Block Generator和Driver上的Job Generator在启动时，都会启动自己的RecurringTimer对象，以定时触发各自的周期性工作。

分布式环境下远程通信是必须的。Spark 1.6推出了RPCEnv、RPCEndpoint、RPCEndpointRef为核心的新型架构下的RPC通信方式。其具体实现有两种方式，Akka和Netty。Akka是基于Scala的Actor的分布式消息通信系统；Netty是一个Java开源框架，提供

异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。目前Spark默认的RPC通信方式是Netty。Spark Streaming中，Driver端的Receiver Tracker、Job Generator以及Executor端的ReceiverSupervisorImpl都会用到这种分布式通信方式。

除了RPC消息循环体，还使用了EventLoop这样的本地消息循环体。有必要对EventLoop做个总结。在生成EventLoop对象时，还会实例化EventLoop对象中的一个消息队列、一个Thread成员。EventLoop获得的消息会放入消息队列中。而这个线程则不断检查消息队列中是否有消息，有则用onReceive方法处理。具体的处理需要在外部实例化该EventLoop时覆盖onReceive方法。Spark Streaming的代码中一般用名为processEvent的函数来对应。Driver的Job Scheduler、Job Generator都使用了EventLoop。

架构中的更多细节以后再深入讨论。

2.4 编程接口

Spark Streaming的编程接口主要由StreamingContext和DStream提供。如果要设计复杂的Spark应用程序，彻底理解StreamingContext和DStream很重要。

StreamingContext类是Spark Streaming的入口类，一定会使用Spark应用程序的入口类SparkContext。

源码2-2 StreamingContext片段

```
class StreamingContext private[streaming] (
    sc_ : SparkContext,
    cp_ : Checkpoint,
    batchDur_ : Duration
```

```
  ) extends Logging {
    ...
}
```

StreamingContext有SparkContext对象、检查点、时间间隔等参数。

流数据的处理是通过InputDStream子类的生成、DStream子类的操作来实现。所以再看看DStream。

源码2-3 DStream片段

```
abstract class DStream[T: ClassTag] (
    @transient private[streaming] var ssc: StreamingContext
  ) extends Serializable with Logging {

  validateAtInit()

  // ========================================================================
  // Methods that should be implemented by subclasses of DStream
  // ========================================================================

  /** Time interval after which the DStream generates a RDD */
  def slideDuration: Duration

  /** List of parent DStreams on which this DStream depends on */
  def dependencies: List[DStream[_]]
```

```
/** Method that generates a RDD for the given time */
def compute(validTime: Time): Option[RDD[T]]
```

从这里可以看出，DStream就是Spark Streaming的核心，就像RDD是Spark Core的核心，它有dependencies和compute。它们体现了DStream的3个关键特点：

（1）除了第一个DStream，后面的DStream都要依赖前面的DStream。

（2）DStream在每一个时间间隔（interval）都会生成一个RDD。

（3）这个类里有一个函数可以在每一个时间间隔后产生一个RDD。

DStream子类会覆写compute。其参数为Time类型。Time类的成员是一个代表时刻的单位为毫秒的长整型数。在Spark Streaming中，Time对象通常用来对应某个批次。

继续看代码。

源码2-4 DStream片段

```
// RDDs generated, marked as private[streaming] so that testsuites can access it
@transient
private[streaming] var generatedRDDs = new HashMap[Time, RDD[T]] ()
```

这里的generatedRDDS是一个HashMap，以时刻为key，以RDD为value。其中的每个RDD实际是该DStream在每个批次所相应生成的RDD。DStream就是RDD的模板。

DStream有很多子类，如图2-11所示。

DStream通过转换（transformation）操作生成新的DStream子类对象，如表2-3所示。

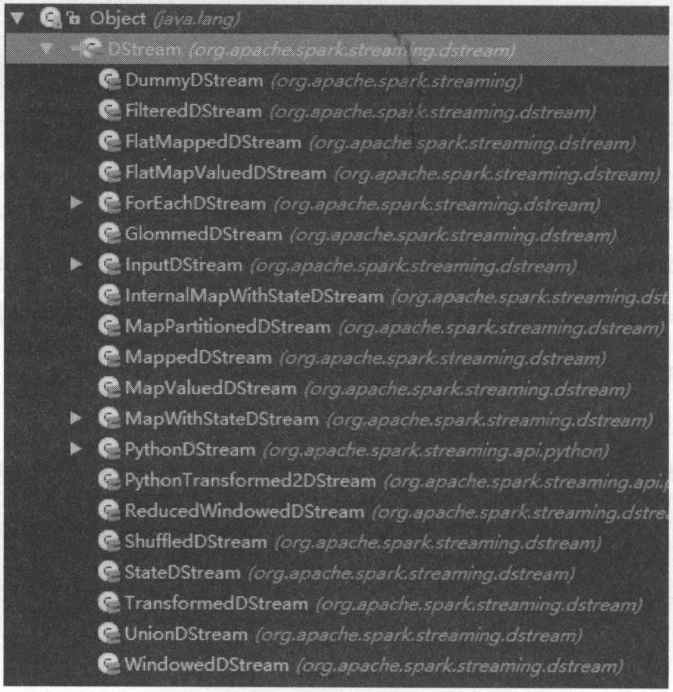


图2-11　DStream及其子类

表2-3　DStream的transformation操作

转换操作	描　述
map(func)	源DStream的每个元素通过函数func返回一个新的DStream
flatMap(func)	类似于map操作，不同的是每个输入元素可以被映射出0个或者更多的输出元素
filter(func)	在源DStream上选择func函数返回仅为true的元素，最终返回一个新的DStream
repartition(numPartitions)	通过输入的参数numPartitions的值来改变DStream的分区大小
union(otherStream)	返回一个包含源DStream与其他 DStream的元素合并后的新DStream
count()	对源DStream内部所含有的RDD的元素数量进行计数，返回一个内部包含的RDD只有一个元素的DStream
reduce(func)	使用函数func（有两个参数并返回一个结果）将源DStream 中每个RDD的元素进行聚合操作，返回一个内部所包含的RDD只有一个元素的新DStream
countByValue()	计算DStream中每个RDD内的元素出现的频次并返回新的DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素出现的频次

续表

转换操作	描 述
reduceByKey(func, [numTasks])	当一个类型为（K，V）键值对的DStream被调用的时候,返回类型为类型为（K，V）键值对的新 DStream,其中每个键K的值V都使用聚合函数func汇总。注意：默认情况下，使用 Spark的默认并行度提交任务（本地模式下并行度为2，集群模式下为8），可以通过配置numTasks设置不同的并行任务数
join(otherStream, [numTasks])	当被调用类型分别为（K，V）和（K，W）键值对的两个DStream时，返回类型为（K，（V，W））键值对的一个新 DStream
cogroup(otherStream, [numTasks])	当被调用的两个DStream分别含有（K，V）和（K，W）键值对时，返回一个（K，Seq[V]，Seq[W]）类型的新的DStream
transform(func)	通过对源DStream的每个RDD应用RDD-to-RDD函数返回一个新的DStream，这可以用来在DStream做任意RDD操作
updateStateByKey(func)	返回一个新状态的DStream，其中每个键的状态是根据键的前一个状态和键的新值应用给定函数func后的更新。这个方法可以用来维持每个键的任何状态数据

DStream的转换操作实际上是生成DStream子类对象。

DStream的状态操作、窗口操作在相关章节再介绍。

Spark Streaming应用程序在对DStream做若干转换操作后，后面会有输出操作，如表2-4所示。

表2-4 DStream的输出操作

输出操作	描 述
print()	在Driver中打印出DStream中数据的前10个元素
saveAsTextFiles(prefix, [suffix])	将DStream中的内容以文本的形式保存为文本文件，其中每次批处理间隔内产生的文件以prefix-TIME_IN_MS[.suffix]的方式命名
saveAsObjectFiles(prefix, [suffix])	将DStream中的内容按对象序列化并且以SequenceFile的格式保存。其中每次批处理间隔内产生的文件以prefix-TIME_IN_MS[.suffix]的方式命名
saveAsHadoopFiles(prefix, [suffix])	将DStream中的内容以文本的形式保存为Hadoop文件，其中每次批处理间隔内产生的文件以prefix-TIME_IN_MS[.suffix]的方式命名
foreachRDD(func)	最基本的输出操作，将func函数应用于DStream中的RDD上，这个操作会输出数据到外部系统，比如保存RDD到文件或者网络数据库等。需要注意的是，func函数是在运行该streaming应用的Driver进程里执行的

DStream的输出操作会生成DStream子类ForeachDStream的对象。

此后，在StreamingContext启动时，会利用这些DStream子类对象来触发RDD的生成和转换，然后产生若干个Job，在集群上运行。

DStream可以说是逻辑级别的，而RDD是物理级别的，DStream所表达的最终都转换成RDD去实现。前者是更高级别的抽象，后者是底层的实现。DStream实际上就是在时间维度上对RDD集合的封装，DStream与RDD的关系就是随着时间流逝不断地产生RDD，对DStream的操作就是在固定时间上操作RDD。

第3章
Spark Streaming运行流程详解

第2章阐述了Spark Streaming的基本原理，本章则在前面这些基本原理的基础上，深入到源码中，对Spark Streaming中的各主要流程进行剖析。对于主要的流程，会先做一些概要性的阐述，并给出流程图。有些流程会比较长，读者可结合流程图来理清思路。

3.1 从StreamingContext的初始化到启动

Spark Streaming应用程序最开始做的事情一般是初始化StreamingContext，即生成StreamingContext对象。因为StreamingContext是Spark Streaming的入口。

StreamingContext初始化一般会有两个参数，分别是SparkConf对象和批处理时间间隔（Batch Duration）。

以下是一个基本的Spark Streaming应用程序范例。

源码3-1 NetworkWordCount

```
package com.dt.spark.streaming


import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}


object NetworkWordCount {
  def main(args: Array[String]) {
     val sparkConf = new SparkConf().setAppName("NetworkEWordCount")
     // 初始化StreamingContext，即生成StreamingContext对象
    val ssc = new StreamingContext(sparkConf, Seconds(1))
```

```
    val lines = ssc.socketTextStream("Master", 9999)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_+_)
    wordCounts.print()
    // 启动StreamingContext
    ssc.start()
    ssc.awaitTermination()
  }
}
```

为突出本节重点，先给出省略了业务处理部分的NetworkWordCount的流程图，如图3-1所示。

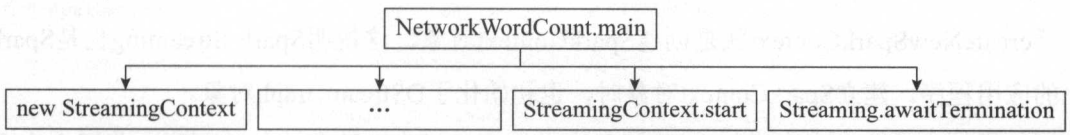


图3-1　NetworkWordCount流程图（有省略部分）

了解Spark基本应用的读者都知道，Spark应用程序都会用到SparkContext类。为何在Spark Streaming应用程序中没看到SparkContext对象呢？其实是因为使用StreamingContext，从而间接使用了SparkContext。StreamingContext的源码如下所示。

源码3-2　StreamingContext片段

```
/**
 * Create a StreamingContext by providing the configuration necessary for a new SparkContext.
 * @param conf a org.apache.spark.SparkConf object specifying Spark parameters
 * @param batchDuration the time interval at which streaming data will be divided into batches
```

```
 */
def this(conf: SparkConf, batchDuration: Duration) = {
  this(StreamingContext.createNewSparkContext(conf), null, batchDuration)
}
```

初始化StreamingContext时，调用了createNewSparkContext。

源码3-3 StreamingContext.createNewSparkContext

```
private[streaming] def createNewSparkContext(conf: SparkConf): SparkContext = {
  new SparkContext(conf)
  }
```

createNewSparkContext就是创建SparkContext对象。这说明Spark Streaming也是Spark上的应用程序。建立SparkContext对象时，也初始化了DStreamGraph对象。

流程图中的省略号部分是应用程序的业务处理部分。随着业务的不同，代码也会不同。其中主要是先创建InputDStream对象，然后是各种DStream的转换操作，最后再输出。这里说的输出，可以是控制台显示，也可以是写到文件中，还可以是发送到其他远程机器上。

以下是NetworkWordCount中的DStream相关代码：

```
val lines = ssc.socketTextStream("Master", 9999)                    // 生成InputDStream对象
val words = lines.flatMap(_.split(" "))                             // DStream的转换操作flatMap
val wordCounts = words.map(x => (x, 1)).reduceByKey(_+_)            // 连续两个DStream转换操作
wordCounts.print()                                                  // DStream输出操作
```

Spark Streaming应用程序的开始部分都会通过数据流创建一个InputDStream。本例中有一行代码：

```
val lines = ssc.socketTextStream("Master", 9999)
```

其中的socketTextStream如下所示。

源码3-4　StreamingContext.socketTextStream

```
/**
 * Create a input stream from TCP source hostname:port. Data is received using
 * a TCP socket and the receive bytes is interpreted as UTF8 encoded ' \n'  delimited
 * lines.
 * @param hostname      Hostname to connect to for receiving data
 * @param port          Port to connect to for receiving data
 * @param storageLevel  Storage level to use for storing the received objects
 *                      (default: StorageLevel.MEMORY_AND_DISK_SER_2)
 */
def socketTextStream(
    hostname: String,
    port: Int,
    storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
  ): ReceiverInputDStream[String] = withNamedScope("socket text stream") {
  socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel)
}
```

可看到代码最后面调用了socketStream。

源码3-5 StreamingContext.socketStream

```
/**
 * Create a input stream from TCP source hostname:port. Data is received using
 * a TCP socket and the receive bytes it interepreted as object using the given
 * converter.
 * @param hostname      Hostname to connect to for receiving data
 * @param port          Port to connect to for receiving data
 * @param converter     Function to convert the byte stream to objects
 * @param storageLevel  Storage level to use for storing the received objects
 * @tparam T            Type of the objects received (after converting bytes to objects)
 */

def socketStream[T: ClassTag](
    hostname: String,
    port: Int,
    converter: (InputStream) => Iterator[T],
    storageLevel: StorageLevel
  ): ReceiverInputDStream[T] = {
  new SocketInputDStream[T](this, hostname, port, converter, storageLevel)
}
```

SocketInputDStream继承ReceiverInputDStream。

其中实现了getReceiver方法，返回SocketReceiver对象。

总结一下SocketInputDStream的继承关系：SocketInputDStream→ReceiverInputDStream→InputDStream→DStream。

DStream是生成RDD的模板，是逻辑级别，当达到Interval的时候这些模板会被实例化为RDD和DAG。

第一行是生成InputDStream对象，该对象会加入到DStreamGraph的成员InputStreams中。接下来的DStream转换操作实际就是生成新的DStream子类对象。

DStream转换操作生成的新的DStream子类对象中会有以下成员：

- 依赖的DStream对象。
- 依赖的DStream对象的转换操作的函数参数。该函数尚未被调用执行。

最后的输出操作会生成ForeachDStream对象，并注册到DStreamGraph的成员outputStreams中。各DStream对象中的依赖信息和操作函数信息为DStreamGraph利用outputStreams进行回溯并生成Job创造了条件。要到StreamingContext启动的时候，这些业务处理中的操作才会具体落实到底层，即对Spark RDD的操作。

现在对DStream的输出操作进行分析。目前已有的DStream的输出操作有print、saveAsTextFiles、saveAsObjectFiles、saveAsHadoopFiles、foreachRDD。

除开foreachRDD，这些输出操作都会做两件事：

（1）定义一个函数，是针对RDD的相应操作。参数是RDD对象和Time对象。

（2）调用foreachRDD，是生成一个ForeachDStream对象并注册到DStreamGraph中。而前面定义的函数是foreachRDD的一个参数。

可以看出，上述定义的函数还没有被调用执行。在应用程序中这些输出操作只是定义如何把处理结果输出到数据库、文件系统等外部系统中的模板。StreamingContext.start之前，DStream输出操作就已被调用，但此函数并没有被调用执行，只是随ForeachDStream对象注册到DStreamGraph中。以DStream.print为例进行剖析

先给出DStream.print流程图，如图3-2所示。

现在来看看DStream.print是如何把ForeachDStream对象注册到DStreamGraph中的。

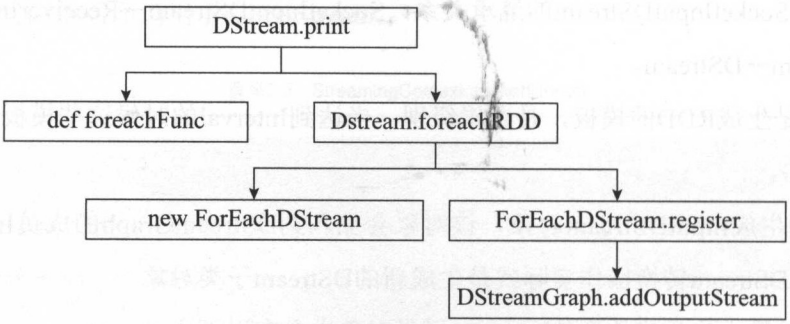


图3-2 DStream的输出操作print的流程

源码3-6 DStream.print

```
/**
 * Print the first ten elements of each RDD generated in this DStream. This is an output
 * operator, so this DStream will be registered as an output stream and there materialized.
 */
def print(): Unit = ssc.withScope {
  print(10)
}
```

其中的print调用了另一个重载的print。

源码3-7 DStream.print

```
/**
 * Print the first num elements of each RDD generated in this DStream. This is an output
 * operator, so this DStream will be registered as an output stream and there materialized.
 */
def print(num: Int): Unit = ssc.withScope {
```

```
  def foreachFunc: (RDD[T], Time) => Unit = {
    (rdd: RDD[T], time: Time) => {
      val firstNum = rdd.take(num + 1)
      // scalastyle:off println
      println("-------------------------------------------")
      println("Time: " + time)
      println("-------------------------------------------")
      firstNum.take(num).foreach(println)
      if (firstNum.length > num) println("...")
      println()
      // scalastyle:on println
    }
  }
  foreachRDD(context.sparkContext.clean(foreachFunc), displayInnerRDDOps = false)
}
```

先定义用于打印输出的函数foreachFunc，该函数没有被执行。然后调用foreachRDD，foreachRDD的参数中有函数foreachFunc。

源码3-8 DStream.foreachRDD

```
/**
 * Apply a function to each RDD in this DStream. This is an output operator, so
 * 'this' DStream will be registered as an output stream and therefore materialized.
 * @param foreachFunc               foreachRDD function
 * @param displayInnerRDDOps        Whether the detailed callsites and scopes of the RDDs generated
```

```
   *                              in the `foreachFunc` to be displayed in the UI. If `false`, then
   *                              only the scopes and callsites of `foreachRDD` will override those
   *                              of the RDDs on the display.
   */
private def foreachRDD(
                          foreachFunc: (RDD[T], Time) => Unit,
                          displayInnerRDDOps: Boolean): Unit = {
    new ForEachDStream(this,
      context.sparkContext.clean(foreachFunc, false), displayInnerRDDOps).register()
  }
```

有新创建的ForeachDStream对象，DStream输出操作中定义的函数被封装在其中，仍没有被执行。这个ForeachDStream对象还调用了register方法。

源码3-9 DStream.register

```
/**
 * Register this streaming as an output stream. This would ensure that RDDs of this
 * DStream will be generated.
 */
private[streaming] def register(): DStream[T] = {
  ssc.graph.addOutputStream(this)
  this
}
```

源码3-10 DStreamGraph.addOutputStream

```
def addOutputStream(outputStream: DStream[_]) {
  this.synchronized {
    outputStream.setGraph(this)
    outputStreams += outputStream
  }
}
```

因此，在注册时，会将当前的ForeachDStream对象加入到graph的outputStreams中。

Spark应用程序一般都不是简单的本地程序，是要在分布式环境下运行的。StreamingContext启动时，需要在分布式集群环境中做好各项准备工作，然后开始数据接收，并且在每个批处理时间把业务处理部分分解成若干个Job，提交到Spark集群中的各个Executor上执行。

下面稍微细化一下StreamingContext启动过程。

首先是数据接收和分配。业务处理部分定义了InputDStream子类，其中一般还会定义具体的流数据接收器（Receiver）。在Executor上用Receiver接收流数据，然后缓存下来，积累成块（Block），再分配给相应流处理时间间隔内的Job。在Driver端需要有个跟踪器（Tracker），这个跟踪器，不仅督促远端的Executor启动Receiver，还要管理待分配给Job的数据Block的元数据。这个跟踪器叫ReceiverTracker。

其次是定期不断生成数据处理Job。像NetworkWordCount这样的程序中定义了初始的InputDStream及其后续的操作，需要转化为相应的RDD，然后根据流处理时间间隔不断生成Job，并提交到Executor上执行。这项工作需要在Driver上有个Job的生成器（Generator）来负责，这个生成器叫JobGenerator。

StreamingContext正常启动的主流程图（有省略部分）如图3-3所示。

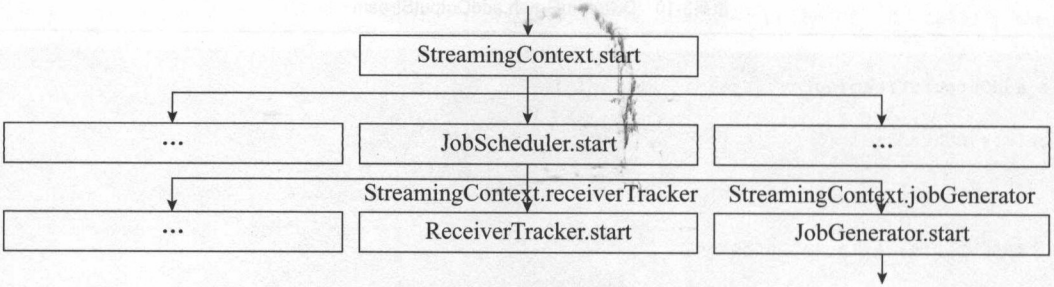


图3-3 StreamingContext正常启动的主流程图

流程图中有ReceiverTracker和JobGenerator的启动。

从StreamingContext的启动开始分析。

源码3-11 StreamingContext.start

```
/**
 * Start the execution of the streams.
 *
 * @throws IllegalStateException if the StreamingContext is already stopped.
 */
def start(): Unit = synchronized {
  state match {
    case INITIALIZED =>
      startSite.set(DStream.getCreationSite())
      StreamingContext.ACTIVATION_LOCK.synchronized {
        StreamingContext.assertNoOtherContextIsActive()
        try {
          validate()

          // Start the streaming scheduler in a new thread, so that thread local properties
```

```
      // like call sites and job groups can be reset without affecting those of the
      // current thread.
      ThreadUtils.runInNewThread("streaming-start") {
        sparkContext.setCallSite(startSite.get)
        sparkContext.clearJobGroup()
        sparkContext.setLocalProperty(SparkContext.SPARK_JOB_INTERRUPT_ON_CANCEL, "false")
        scheduler.start()
      }
      state = StreamingContextState.ACTIVE
    } catch {
      case NonFatal(e) =>
        logError("Error starting the context, marking it as stopped", e)
        scheduler.stop(false)
        state = StreamingContextState.STOPPED
        throw e
    }
    StreamingContext.setActiveContext(this)
  }
  shutdownHookRef = ShutdownHookManager.addShutdownHook(
    StreamingContext.SHUTDOWN_HOOK_PRIORITY)(stopOnShutdown)
  // Registering Streaming Metrics at the start of the StreamingContext
    assert(env.metricsSystem != null)
  env.metricsSystem.registerSource(streamingSource)
  uiTab.foreach(_.attach())
  logInfo("StreamingContext started")
case ACTIVE =>
```

```
      logWarning("StreamingContext has already been started")
    case STOPPED =>
      throw new IllegalStateException("StreamingContext has already been stopped")
  }
}
```

最开始的state值为INITIALIZED，所以在一个线程中启动了JobScheduler对象。

源码3-12 JobScheduler.start

```
def start(): Unit = synchronized {
  if (eventLoop != null) return // scheduler has already been started

  logDebug("Starting JobScheduler")
  eventLoop = new EventLoop[JobSchedulerEvent]("JobScheduler") {
    override protected def onReceive(event: JobSchedulerEvent): Unit = processEvent(event)

    override protected def onError(e: Throwable): Unit = reportError("Error in job scheduler", e)
  }
  // 启动消息循环处理线程，用于处理JobScheduler的各种事件
  eventLoop.start()

  // attach rate controllers of input streams to receive batch completion updates
  for {
    inputDStream <- ssc.graph.getInputStreams
    rateController <- inputDStream.rateController
  } ssc.addStreamingListener(rateController)
```

```
  // 启动监听器，用于更新Spark UI中StreamTab的内容
  listenerBus.start(ssc.sparkContext)
  receiverTracker = new ReceiverTracker(ssc)
  // 生成InputInfoTracker，用于管理所有的输入的流以及输入的数据统计
  // 这些信息将通过StreamingListener监听
  inputInfoTracker = new InputInfoTracker(ssc)
  // 启动ReceiverTracker，用于处理数据接收、数据缓存、Block生成
  receiverTracker.start()
  // 启动JobGenerator
  // 用于DStreamGraph初始化、DStream与RDD的转换、生成Job、提交执行等工作
  jobGenerator.start()
  logInfo("Started JobScheduler")
}
```

JobScheduler对象启动时，启动了EventLoop和StreamingListenerBus，还生成了InputInfoTracker。然后启动ReceiverTracker和JobGenerator这两个重要的对象。这两个对象分别对应数据接收、数据处理工作，在后面会着重剖析。

回到NetworkWordCount（源码3-1）的最后部分。StreamingContext启动之后，会执行ssc.awaitTermination。

源码3-13 StreamingContext.awaitTermination

```
/**
 * Wait for the execution to stop. Any exceptions that occurs during the execution
 * will be thrown in this thread.
 */
```

```
def awaitTermination() {
  waiter.waitForStopOrError()
}
```

就是调用ContextWaiter. waitForStopOrError，其主要作用就是等待执行停止或出现错误，否则应用程序就不会结束，一般情况下会一直运行着。

3.2 数据接收

数据接收主要有以下事情：在Executor上启动Receiver以持续接收流数据。接收的数据是离散的，必须收集成Block，让BlockManager进行保存，并通知Driver端已保存的Block的元数据信息。

先剖析一下Receiver类。

源码3-14 Receiver类片段

```
/**
 * :: DeveloperApi ::
 * Abstract class of a receiver that can be run on worker nodes to receive external data. A
 * custom receiver can be defined by defining the functions `onStart()` and `onStop()`. `onStart()`
 * should define the setup steps necessary to start receiving data,
 * and `onStop()` should define the cleanup steps necessary to stop receiving data.
 * Exceptions while receiving can be handled either by restarting the receiver with `restart(...)`
 * or stopped completely by `stop(...)` or
 *
```

```
 * A custom receiver in Scala would look like this.
 *
 * {{{
 *   class MyReceiver(storageLevel: StorageLevel) extends NetworkReceiver [String](storageLevel) {
 *       def onStart() {
 *           // Setup stuff (start threads, open sockets, etc.) to start receiving data.
 *           // Must start new thread to receive data, as onStart() must be non-blocking.
 *
 *           // Call store(...) in those threads to store received data into Spark's memory.
 *
 *           // Call stop(...), restart(...) or reportError(...) on any thread based on how
 *           // different errors needs to be handled.
 *
 *           // See corresponding method documentation for more details
 *       }
 *
 *       def onStop() {
 *           // Cleanup stuff (stop threads, close sockets, etc.) to stop receiving data.
 *       }
 *   }
 * }}}
 *
 ...
 */
@DeveloperApi
```

```
abstract class Receiver[T](val storageLevel: StorageLevel) extends Serializable {
...
```

有一大段注释，解释了如何定做自己的Receiver。Receiver是抽象类，而且是Serializable子类。因为Receiver是要通过Driver序列化后传到Executor上去接收数据。

Spark中已有的Receiver子类如图3-4所示。

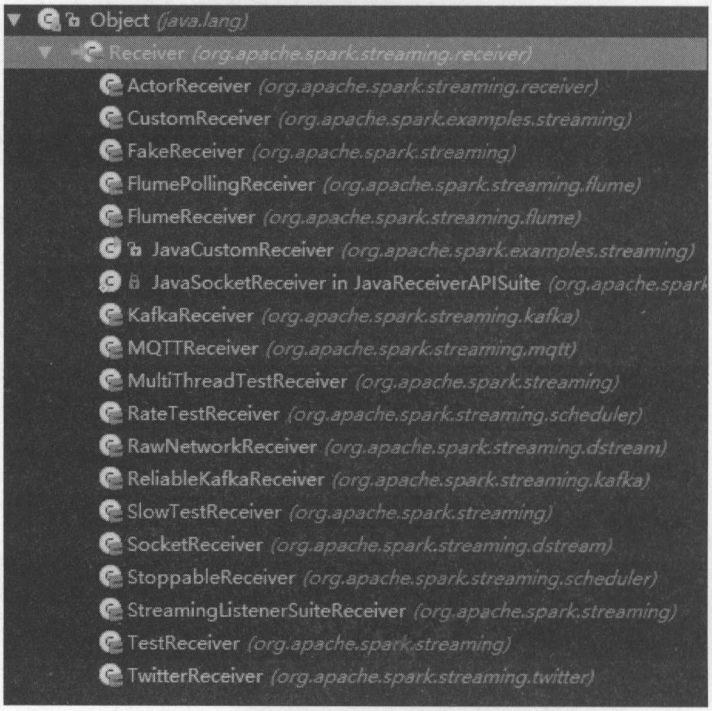


图3-4　Receiver及其子类

Executor端的数据接收由Driver端的ReceiverTracker统一管理。

至于ReceiverTracker启动的流程图，由于流程比较长，先看前一部分，如图3-5所示。

图中SparkContext.submitJob处粗箭头下面的流程是在Executor上运行的，其他流程是在Driver上运行的。

ReceiverTracker.start的源码如下所示。

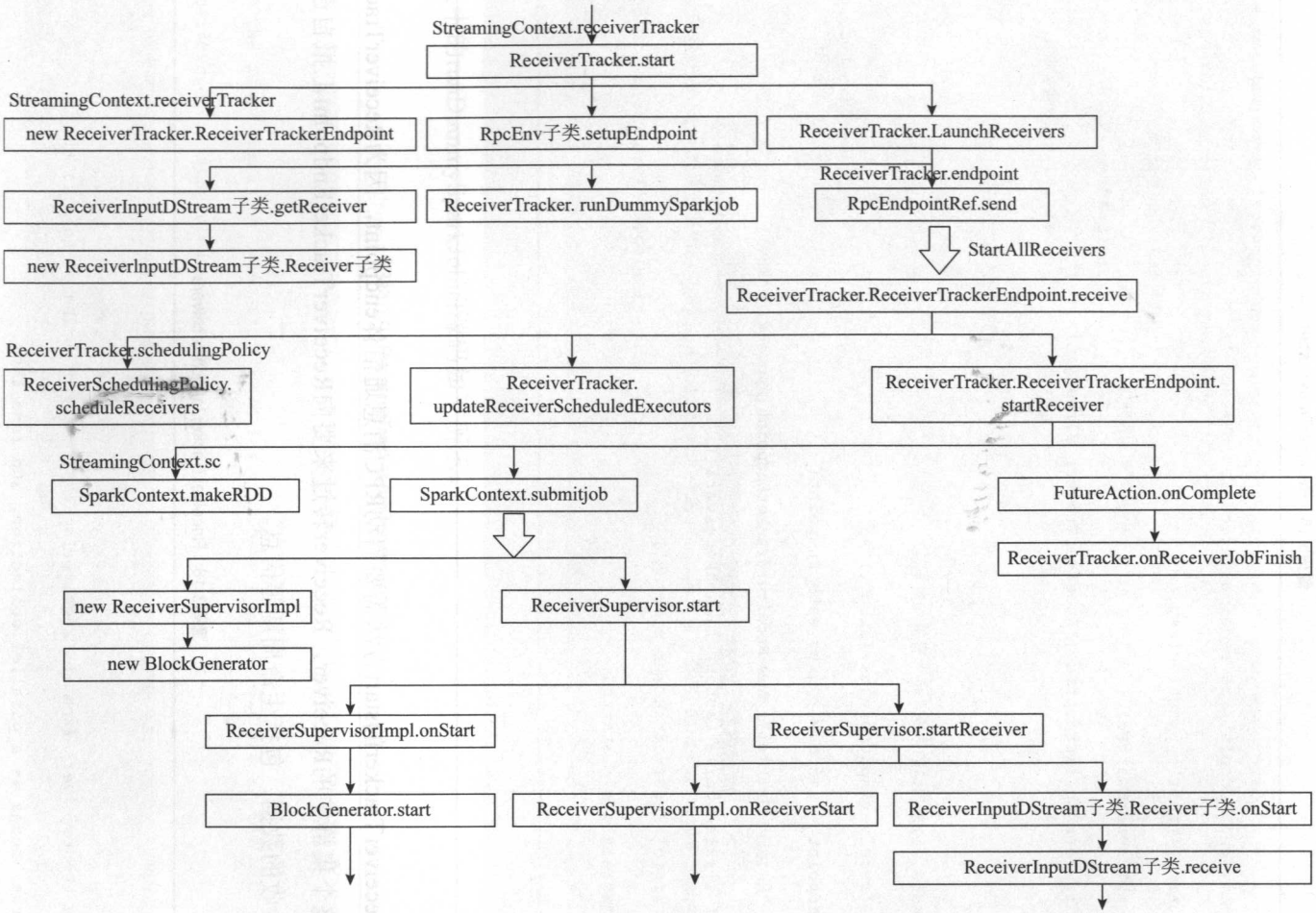


图3-5　ReceiverTracker正常启动的主流程图（前一部分）

源码3-15 ReceiverTracker.start

```
/** Start the endpoint and receiver execution thread. */
def start(): Unit = synchronized {
  if (isTrackerStarted) {
    throw new SparkException("ReceiverTracker already started")
  }

  // Receiver的启动的依据：是否有输入数据流
  if (!receiverInputStreams.isEmpty) {
    endpoint = ssc.env.rpcEnv.setupEndpoint(
      "ReceiverTracker", new ReceiverTrackerEndpoint(ssc.env.rpcEnv))
    if (!skipReceiverLaunch) launchReceivers()
    logInfo("ReceiverTracker started")
    trackerState = Started
  }
}
```

ReceiverTracker的start方法需要启动RPC消息通信体endpoint，因为ReceiverTracker会监控整个集群中的Receiver，Receiver转过来要向ReceiverTrackerEndpoint汇报自己的状态、接收的数据，包括生命周期等信息。

源码3-16 ReceiverTracker.launchReceivers

```
/**
 * Get the receivers from the ReceiverInputDStreams, distributes them to the
 * worker nodes as a parallel collection, and runs them.
```

```
 */
private def launchReceivers(): Unit = {
  val receivers = receiverInputStreams.map(nis => {
    // 一个ReceiverInputDStream只对应一个Receiver
    val rcvr = nis.getReceiver()
    rcvr.setReceiverId(nis.id)
    rcvr
  })

  runDummySparkJob()

  logInfo("Starting " + receivers.length + " receivers")
  // 此时的endpoint就是上面代码中在ReceiverTracker的start方法中构造的ReceiverTrackerEndpoint
  endpoint.send(StartAllReceivers(receivers))
}
```

其中的runDummySparkJob的源码如下所示。

源码3-17　ReceiverTracker.runDummySparkJob

```
/**
 * Run the dummy Spark job to ensure that all slaves have registered. This avoids all the
 * receivers to be scheduled on the same node.
 *
 * TODO Should poll the executor number and wait for executors according to
 * "spark.scheduler.minRegisteredResourcesRatio"and
```

```
 * "spark.scheduler.maxRegisteredResourcesWaitingTime" rather than running a dummy job.
 */
private def runDummySparkJob(): Unit = {
  if (!ssc.sparkContext.isLocal) {
    ssc.sparkContext.makeRDD(1 to 50, 50).map(x => (x, 1)).reduceByKey(_ + _, 20).collect()
  }
  assert(getExecutors.nonEmpty)
}
```

其中利用makeRDD生成一个RDD，然后进行map、reduceByKey、collect操作。collect是action操作，会触发Spark Job的执行。ReceiverTracker.runDummySparkJob就是通过运行一个简单的作业来确保所有的slave节点都已经注册。这样就意味着所有节点活着，在后面分配Receivers时，可以避免所有的Receivers集中在一个节点上。注释中流露出想改变这种运行虚拟Job的做法。

其中的collect调用了SparkContext.runJob，其中先后调用了RDD.partitions、ParallelCollectionRDD.getPartitions、ParallelCollectionRDD.slice，将一个集合切分成多个子集合，这样做可以让多个Receiver有效地运行在Spark上。这里不一一列出这些Spark Core的源码。

再回去看ReceiverTracker.launchReceivers（源码3-16）中的getReceiver()。

源码3-18 ReceiverInputDStream.getReceiver

```
/**
 * Gets the receiver object that will be sent to the worker nodes
 * to receive data. This method needs to defined by any specific implementation
 * of a ReceiverInputDStream.
```

```
 */
def getReceiver(): Receiver[T] // 返回的是Receiver对象
```

ReceiverInputDStream的getReceiver方法返回Receiver对象。 该方法实际上要靠ReceiverInputDStream的子类实现。

ReceiverInputDStream的子类还必须定义自己对应的Receiver子类，因为这个Receiver子类会在getReceiver方法中用来创建这个Receiver子类的对象。了解ReceiverInputDStream子类及其相应的Receiver子类很重要，因为对于特别的数据源，往往需要开发者自己去定制。

下面以ReceiverInputDStream的子类SocketInputDStream为例进行分析，如图3-6所示。

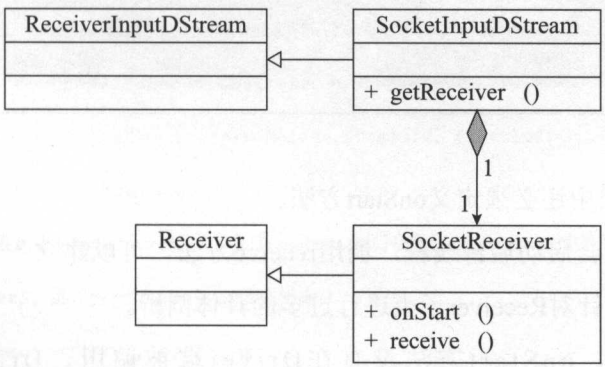


图3-6　SocketInputDStream相关类图

根据继承关系，这里看一下SocketInputDStream中的getReceiver方法。

源码3-19　SocketInputDStream.getReceiver

```
def getReceiver(): Receiver[T] = {
    new SocketReceiver(host, port, bytesToObjects, storageLevel)
  }
}
```

SocketInputDStream中还定义了相应的Receiver子类SocketReceiver。

源码3-20 SocketReceiver片段

```
private[streaming]
class SocketReceiver[T: ClassTag](
      host: String,
      port: Int,
      bytesToObjects: InputStream => Iterator[T],
      storageLevel: StorageLevel
  )   extends Receiver[T](storageLevel) with Logging {
  ...
}
```

SocketReceiver类中还必须定义onStart方法。

这个onStart方法会启动后台线程，调用receive方法，可以建立一个Socket连接，并持续接收数据。这里不针对Receiver子类进行过多的具体剖析。

需要注意的是，onStart方法没有在Driver端被调用。Driver端只是生成了Receiver对象，但Receiver对象并不在Driver端工作。Receiver是可序列化的类，其对象会在Executor上反序列化生成，稍后在介绍Executor工作时可以看到，在ReceiverSupervisorImpl.onReceiverStart（源码3-25）中调用这个onStart方法后，Receiver对象才开始工作。

再回到ReceiverTracker.launchReceivers()中，可以发现endpoint.send(StartAllReceivers(receivers))。ReceiverTrackerEndpoint对象发送了StartAllReceivers消息，这个消息中的参数是所有新Receiver。ReceiverTrackerEndpoint对象接收后所做的处理在ReceiverTrackerEndpoint.receive中。

源码3-21 ReceiverTracker.ReceiverTrackerEndpoint.receive

```
override def receive: PartialFunction[Any, Unit] = {
  // Local messages
  case StartAllReceivers(receivers) =>
    // schedulingPolicy调度策略
    // receivers就是要启动的Receiver
    // getExecutors获得集群中的Executors的列表
    // scheduleReceivers就可以确定receiver可以运行在哪些Executor上
    val scheduledLocations = schedulingPolicy.scheduleReceivers(receivers, getExecutors)
    for (receiver <- receivers) {
    // scheduledLocations根据receiver的Id就能确定当前哪些Executors可以运行Receiver
    val executors = scheduledLocations(receiver.streamId)
      updateReceiverScheduledExecutors(receiver.streamId, executors)
      receiverPreferredLocations(receiver.streamId) = receiver.preferredLocation
    // 上述代码之后要启动的Receiver确定了，具体Receiver运行在哪些Executors上也确定了
    // 循环receivers，每次将一个Receiver传入过去
      startReceiver(receiver, executors)
    }
  case RestartReceiver(receiver) =>
    ⋮
  case c: CleanupOldBlocks =>
    ⋮
  case UpdateReceiverRateLimit(streamUID, newRate) =>
    ⋮
}
```

ReceiverSchedulingPolicy.scheduleReceivers确定receiver可以运行在哪些Executor上，然后用startReceiver循环启动各Receiver。

源码3-22 ReceiverTracker.ReceiverTrackerEndpoint.startReceiver

```
/**
 * Start a receiver along with its scheduled executors
 */
private def startReceiver(
    receiver: Receiver[_],
    // scheduledLocations指定具体在哪台物理机器上执行
    scheduledLocations: Seq[TaskLocation]): Unit = {
    // 判断Receiver的状态是否正常
  def shouldStartReceiver: Boolean = {
    // It's okay to start when trackerState is Initialized or Started
    !(isTrackerStopping || isTrackerStopped)
  }

  val receiverId = receiver.streamId
  if (!shouldStartReceiver) {
    // 如果不需要启动Receiver则会调用
    onReceiverJobFinish(receiverId)
    return
  }

  val checkpointDirOption = Option(ssc.checkpointDir)
```

```
val serializableHadoopConf =
  new SerializableConfiguration(ssc.sparkContext.hadoopConfiguration)

// startReceiverFunc封装了在Worker上启动Receiver的动作
// Function to start the receiver on the worker node
val startReceiverFunc: Iterator[Receiver[_]] => Unit =
  (iterator: Iterator[Receiver[_]]) => {
    if (!iterator.hasNext) {
      throw new SparkException(
        "Could not start receiver as object not found.")
    }
    if (TaskContext.get().attemptNumber() == 0) {
      val receiver = iterator.next()
      assert(iterator.hasNext == false)
      // ReceiverSupervisorImpl是Receiver的监控器，同时负责数据的写等操作
      val supervisor = new ReceiverSupervisorImpl(
        receiver, SparkEnv.get, serializableHadoopConf.value, checkpointDirOption)
      supervisor.start()
      supervisor.awaitTermination()
    } else {
      // 若想重启Receiver，则需重新完成上面的调度，而不是Task重试
      // It's restarted by TaskScheduler, but we want to reschedule it again. So exit it.
    }
  }
```

```
// 生成Receiver的RDD，此RDD用于该Receiver在Worker上在生成和启动
// Create the RDD using the scheduledLocations to run the receiver in a Spark job
val receiverRDD: RDD[Receiver[_]] =
  if (scheduledLocations.isEmpty) {
    ssc.sc.makeRDD(Seq(receiver), 1)
  } else {
    val preferredLocations = scheduledLocations.map(_.toString).distinct
    ssc.sc.makeRDD(Seq(receiver -> preferredLocations))
  }
// 从 receiverId可以看出，Receiver只有一个
receiverRDD.setName(s"Receiver $receiverId")
ssc.sparkContext.setJobDescription(s"Streaming job running receiver $receiverId")
ssc.sparkContext.setCallSite(Option(ssc.getStartSite()).getOrElse(Utils.getCallSite()))

// 每个Receiver的启动都会触发一个Job，而不是一个作业的Task去启动所有的Receiver
// 应用程序可能有若干个Receiver
// 调用SparkContext的submitJob，为了启动Receiver，启动了一个Spark作业
val future = ssc.sparkContext.submitJob[Receiver[_], Unit, Unit](
  receiverRDD, startReceiverFunc, Seq(0), (_, _) => Unit, ())
// We will keep restarting the receiver job until ReceiverTracker is stopped
future.onComplete {
  case Success(_) =>
    if (!shouldStartReceiver) {
      onReceiverJobFinish(receiverId)
    } else {
      logInfo(s"Restarting Receiver $receiverId")
```

```
        self.send(RestartReceiver(receiver))
      }
    case Failure(e) =>
      if (!shouldStartReceiver) {
        onReceiverJobFinish(receiverId)
      } else {
        logError("Receiver has been stopped. Try to restart it.", e)
        logInfo(s"Restarting Receiver $receiverId")
        self.send(RestartReceiver(receiver))
      }
  // 使用线程池的方式提交Job，这样的好处是可以并发地启动Receiver
  }(submitJobThreadPool)
  logInfo(s"Receiver ${receiver.streamId} started")
}
```

从注释中可以看到，Spark Streaming指定Receiver在哪些Executors上运行，而不是基于Spark Core中的Task来指定。

函数startReceiverFunc虽然是在submitJob之前定义，但要通过SparkContext.submitJob提交到Executor端去执行。将Receiver转化为Spark能够处理的RDD，此RDD也是SparkContext.submitJob的一个参数。所以Spark Streaming是使用SparkContext.submitJob的方式在Executor上启动一个Receiver。一个Job只启动一个Receiver。

当Receiver启动失败时，就会触发ReceiverTrackEndpoint重新启动一个Spark Job去启动Receiver。

函数在Executor中运行时，会调用ReceiverSupervisor.start。

源码3-23 ReceiverSupervisor.start

```
/** Start the supervisor */
def start() {
  onStart() // 具体实现是通过子类完成的
  startReceiver()
}
```

下面看ReceiverSupervisor子类ReceiverSupervisorImpl中的onStart。

源码3-24 ReceiverSupervisorImpl.onStart

```
overrideprotected def onStart() {
  registeredBlockGenerators.foreach { _.start() }
}
```

这实际上是启动其中的每一个BlockGenerator。其中的_.start()是BlockGenerator.start。这是启动数据Block的生成器。按照流程图暂且剖析到这里。

回到ReceiverSupervisor.start（源码3-23），再看startReceiver。

源码3-25 ReceiverSupervisor.startReceiver

```
/** Start receiver */
def startReceiver(): Unit = synchronized {
  try {
    if (onReceiverStart()) {
```

```
      logInfo("Starting receiver")
      receiverState = Started
      receiver.onStart()
      logInfo("Called receiver onStart")
    }else {
      // The driver refused us
      stop("Registered unsuccessfully because Driver refused to start receiver "+ streamId, None)
    }
  } catch {
    case NonFatal(t) =>
      stop("Error starting receiver " + streamId, Some(t))
  }
}
```

正常流程情况下，会满足条件ReceiverSupervisorImpl.onReceiverStart，然后调用receiver.onStart。这个receiver是ReceiverInputDStream子类中定义的Receiver子类，是由Spark Streaming应用程序设置流数据接收时所关联的InputDStream决定的。

至此剖析完了图3-5的流程。下面进一步剖析其后续的各子流程。

先分析其中的ReceiverSuperImpl.onReceiverStart。ReceiverSupervisorImpl.onReceiverStart主要用来更新UITab相关的Receiver状态信息。其流程与其他子流程相比稍显独立。

运行在Executor端的ReceiverSupervisorImpl需要与Driver端的ReceiverTracker进行通信，传递元数据信息。ReceiverSupervisorImpl是通过RPC的名称获取ReceiverTracker的远程调用。

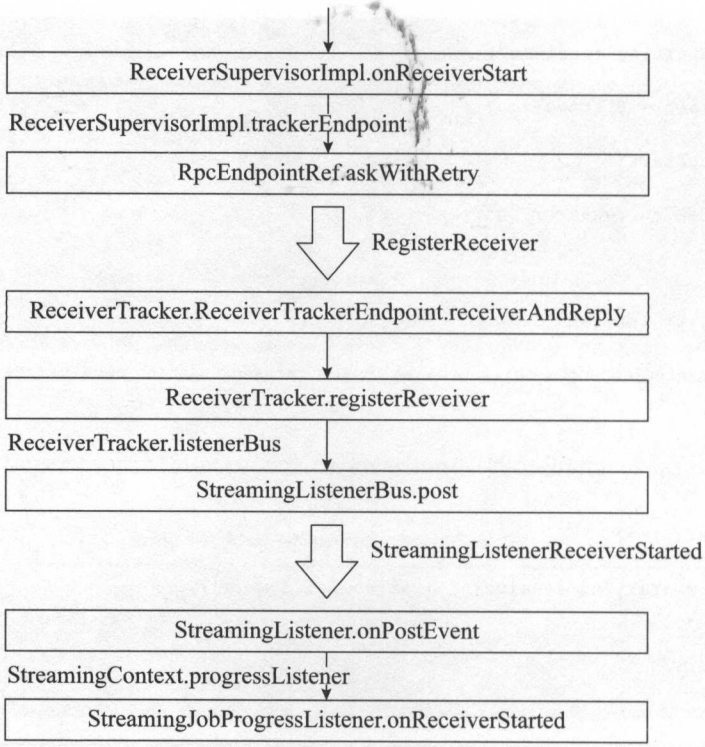


图3-7 ReceiverTracker正常启动的主流程图（后部分之一）：ReceiverSupervisorImpl.onReceiverStart

源码3-26 ReceiverSupervisorImpl.onReceiverStart

```
override protected def onReceiverStart(): Boolean = {
  val msg = RegisterReceiver(
    streamId, receiver.getClass.getSimpleName, host, executorId, endpoint)
  trackerEndpoint.askWithRetry[Boolean](msg)
}
```

这个trackerEndpoint是远端的RpcEndpointRef。所以这里是向Driver端的ReceiverTracker发送RegisterReceiver消息。针对Executor端发送来的消息，Driver端的

ReceiverTracker是使用ReceiverTrackerEndpoint的receiveAndReply来接收。

源码3-27 ReceiverTracker.ReceiverTrackerEndpoint.receiveAndReply

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  // Remote messages
  case RegisterReceiver(streamId, typ, host, executorId, receiverEndpoint) =>
    val successful =
      registerReceiver(streamId, typ, host, executorId, receiverEndpoint, context.senderAddress)
    context.reply(successful)
  case AddBlock(receivedBlockInfo) =>
    ...
  case DeregisterReceiver(streamId, message, error) =>
    ...
  case AllReceiverIds =>
    ...
  case StopAllReceivers =>
    ...
}
```

receiveAndReply调用了registerReceiver。

源码3-28 ReceiverTracker.registerReceiver

```
/** Register a receiver */
private def registerReceiver(
    streamId: Int,
```

```
    typ: String,
    host: String,
    executorId: String,
    receiverEndpoint: RpcEndpointRef,
    senderAddress: RpcAddress
  ): Boolean = {
  if (!receiverInputStreamIds.contains(streamId)) {
    throw new SparkException("Register received for unexpected id "+ streamId)
  }

  if (isTrackerStopping || isTrackerStopped) {
    return false
  }

  val scheduledLocations = receiverTrackingInfos(streamId).scheduledLocations
  val acceptableExecutors = if (scheduledLocations.nonEmpty) {
      // This receiver is registering and it's scheduled by
      // ReceiverSchedulingPolicy.scheduleReceivers. So use "scheduledLocations" to check it.
      scheduledLocations.get
    } else {
      // This receiver is scheduled by "ReceiverSchedulingPolicy.rescheduleReceiver", so calling
      // "ReceiverSchedulingPolicy.rescheduleReceiver" again to check it.
      scheduleReceiver(streamId)
    }

  def isAcceptable: Boolean = acceptableExecutors.exists {
```

```
    case loc: ExecutorCacheTaskLocation => loc.executorId == executorId
    case loc: TaskLocation => loc.host == host
  }

  if (!isAcceptable) {
    // Refuse it since it's scheduled to a wrong executor
    false
  } else {
    val name = s"${typ}-${streamId}"
    val receiverTrackingInfo = ReceiverTrackingInfo(
      streamId,
      ReceiverState.ACTIVE,
      scheduledLocations = None,
      runningExecutor = Some(ExecutorCacheTaskLocation(host, executorId)),
      name = Some(name),
      endpoint = Some(receiverEndpoint))
    receiverTrackingInfos.put(streamId, receiverTrackingInfo)
    listenerBus.post(StreamingListenerReceiverStarted(receiverTrackingInfo.toReceiverInfo))
    logInfo("Registered receiver for stream "+ streamId + "from "+ senderAddress)
    true
  }
}
```

registerReceiver将该Receiver的跟踪信息放入ReceiverTracker的receiverTrackingInfos中，并且向JobScheduler的listener发送了StreamingListenerReceiverStarted消息。

JobScheduler的listener是StreamingListerBus对象，用onPostEvent来接收信息并处理。

源码3-29 StreamingListerBus.onPostEvent

```
override def onPostEvent(listener: StreamingListener, event: StreamingListenerEvent): Unit = {
  event match {
    case receiverStarted: StreamingListenerReceiverStarted =>
      listener.onReceiverStarted(receiverStarted)
    ...
  }
}
```

StreamingListener的子类StreamingJobProgressListener的对象progressListener是StreamingContext的私有成员，它会做相应处理。

源码3-30 StreamingJobProgressListener.onReceiverStarted

```
override def onReceiverStarted(receiverStarted: StreamingListenerReceiverStarted) {
  synchronized {
    receiverInfos(receiverStarted.receiverInfo.streamId) = receiverStarted.receiverInfo
  }
}
```

StreamingContext的progressListener会更新自己的成员receiverInfos中的相应Receiver的信息。

ReceiverSupervisorImpl的onReceiverStart的流程就剖析完了。

接下来分析数据接收流程中剩余的两个子流程：BlockGenerator.start和ReceiverInputDStream

子类.receive，这些流程负责用接收的数据生成Block，为Job的执行做好数据准备。先给出一个相关的流程图，如图3-8所示。

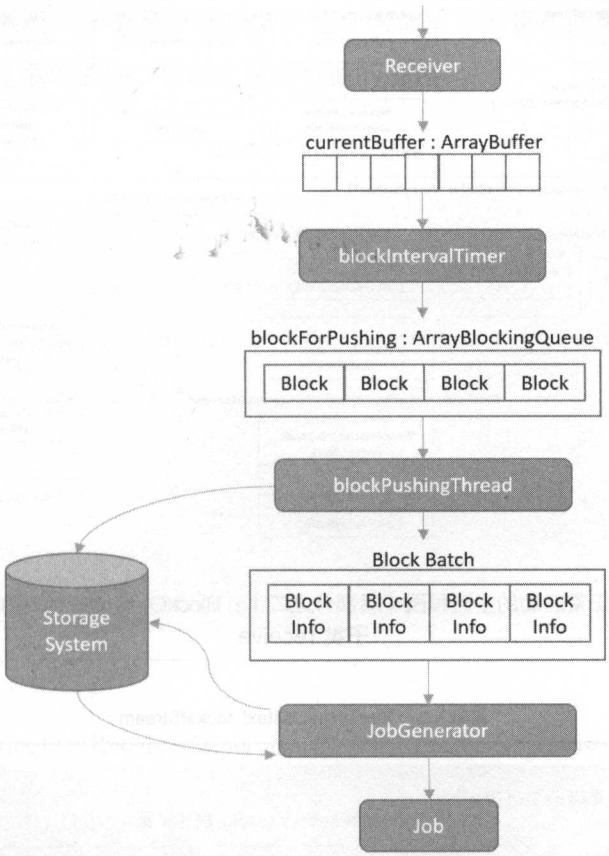


图3-8　数据接收是为数据处理做数据准备

图中的Storage System是用内存或磁盘保存块数据。线程blockIntervalTimer和blockPushingThread是BlockGenarator中的两个成员，在生成Block的环节担负着重要角色。

BlockGenerator.start和ReceiverInputDStream子类.receive这两个流程是有密切联系的，所以流程图也合在一起，如图3-9所示。

先看ReceiverInputDStream子类.receive。不同Spark Streaming应用程序中使用的ReceiverInputDStream子类不一定相同。仍以源码3-1的NetworkWordCount为例进行说明。程序中接收源数据的代码是 val lines = ssc.socketTextStream("Master", 9999)。

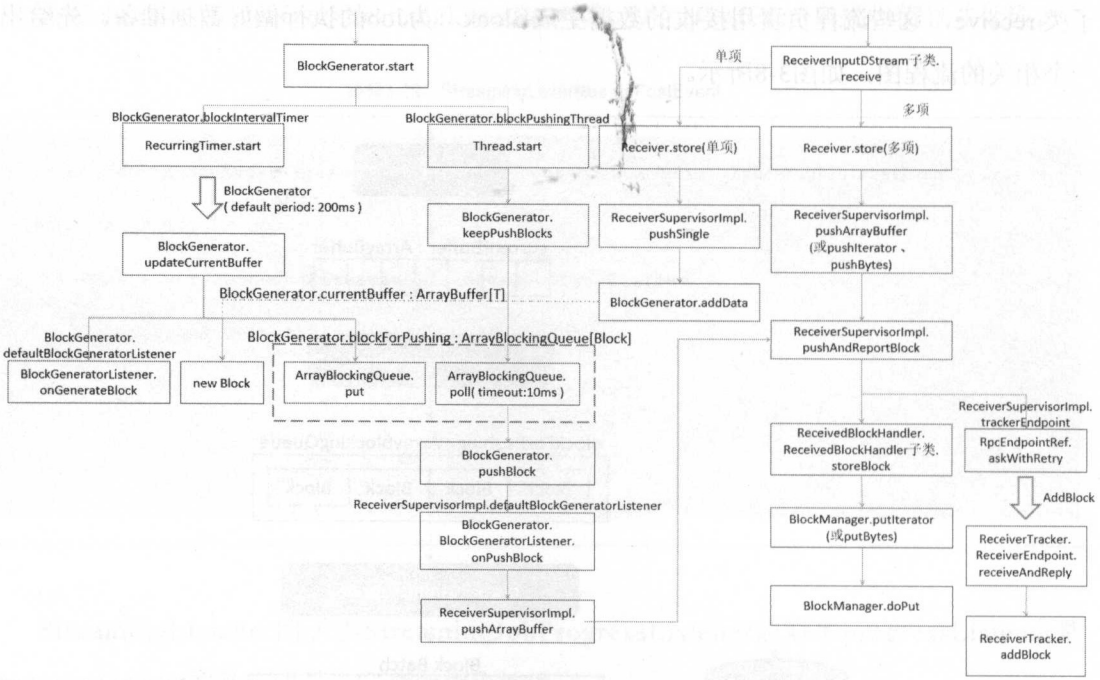


图3-9 ReceiverTracker正常启动的主流程图（后部分之二）：BlockGenerator.start和ReceiverInputDStream子类.receive

源码3-31 StreamingContext.socketStream

```
def socketStream[T: ClassTag](
    hostname: String,
    port: Int,
    converter: (InputStream) => Iterator[T],
    storageLevel: StorageLevel
  ): ReceiverInputDStream[T] = {
  new SocketInputDStream[T](this, hostname, port, converter, storageLevel)
}
```

最终实际会生成一个SocketInputDStream对象。所以NetworkWordCount会使用SocketInputDStream.receive来接收源数据。接收数据后，都会用Receiver.store写入内存。写入内存的方式有两类：一类是单项数据，这样的数据需要累积起来形成Block后，再分配给作业；另一类是多项数据。先剖析单项数据的流程。

源码3-32　Receiver.store

```
/**
 * Store a single item of received data to Spark's memory.
 * These single items will be aggregated together into data blocks before
 * being pushed into Spark's memory.
 */
def store(dataItem: T) {
  supervisor.pushSingle(dataItem)
}
```

被调用的pushSingle在ReceiverSupervisor的子类ReceiverSupervisorImpl中定义。

源码3-33　ReceiverSupervisor.pushSingle

```
/** Push a single record of received data into block generator. */
def pushSingle(data: Any) {
  defaultBlockGenerator.addData(data)
}
```

这里的defaultBlockGenerator是BlockGenerator对象。

源码3-34 BlockGenerator.addData

```
/**
 * Push a single data item into the buffer.
 */
def addData(data: Any): Unit = {
  if (state == Active) {
    waitToPush()
    synchronized {
      if (state == Active) {
        currentBuffer += data
      } else {
        throw new SparkException(
          "Cannot add data as BlockGenerator has not been started or has been stopped")
      }
    }
  } else {
    throw new SparkException(
      "Cannot add data as BlockGenerator has not been started or has been stopped")
  }
}
```

数据会加入ArrayBuffer类型的BlockGenerator成员currentBuffer中。currentBuffer中数据的使用就靠BlockGenerator。

接下来看BlockGenerator的启动流程。

源码3-35 BlockGenerator.start

```
/** Start block generating and pushing threads. */
def start(): Unit = synchronized {
  if (state == Initialized) {
    state = Active
    blockIntervalTimer.start()
    blockPushingThread.start()
    logInfo("Started BlockGenerator")
  } else {
    throw new SparkException(
      s"Cannot start BlockGenerator as its not in the Initialized state [state = $state]")
  }
}
```

BlockGenerator.start启动了两个对象：

（1）RecurringTimer对象blockIntervalTimer。

（2）Thread对象blockPushingThread。

这两个对象对应BlockGenerator内部的两个线程：

（1）第一个线程需要周期性地把先前接收的数据封装成Block。所以此线程放在了定时器类RecurringTimer中。

（2）第二个线程blockPushingThread把Block写入到BlockManager中。

先看RecurringTimer对象blockIntervalTimer。

源码3-36 BlockGenerator.blockIntervalTimer

```
private val blockIntervalTimer =
```

```
new RecurringTimer(clock, blockIntervalMs, updateCurrentBuffer, "BlockGenerator")
```

启动blockIntervalTimer后，会周期性地运行updateCurrentBuffer。blockIntervalTimer的定时周期是blockIntervalMs，其默认是200ms，可用spark.streaming.blockInterval参数进行配置。

源码3-37 BlockGenerator.updateCurrentBuffer

```
/** Change the buffer to which single records are added to. */
private def updateCurrentBuffer(time: Long): Unit = {
  try {
    var newBlock: Block = null
    // 不同线程都会访问currentBuffer，故需加锁
    synchronized {
      // 如果缓冲器不为空，则生成StreamBlockId对象
      // 调用listener的onGenerateBlock来通知Block已生成
      // 再实例化block对象
      if (currentBuffer.nonEmpty) {
        val newBlockBuffer = currentBuffer
        currentBuffer = new ArrayBuffer[Any]
        // 生成新的StreamBlockId: "input-" + streamId + "-"+ (调用time-blockInterval)
        val blockId = StreamBlockId(receiverId, time - blockIntervalMs)
        listener.onGenerateBlock(blockId)
        // 把当前StreamBlockId和currentBuffer封装成Block
        newBlock = new Block(blockId, newBlockBuffer)
      }
```

```
    }

    if (newBlock != null) {
      // 把新建的Block放入blocksForPushing队列中
      blocksForPushing.put(newBlock)  // put is blocking when queue is full
    }
  } catch {
    case ie: InterruptedException =>
      logInfo("Block updating timer thread was interrupted")
    case e: Exception =>
      reportError("Error in block updating thread", e)
  }
}
```

每个定时周期里所做的是，在BlockGenerator的currentBuffer中有数据的情况下，将数据生成的Block对象加入到ArrayBlockingQueue[Block]类型的队列blocksForPushing中。

再看第二个线程blockPushingThread。

源码3-38 BlockGenerator.blockPushingThread

```
private val blockPushingThread = new Thread() { override def run() { keepPushingBlocks() } }
```

源码3-39 BlockGenerator.keepPushingBlocks

```
/** Keep pushing blocks to the BlockManager. */
private def keepPushingBlocks() {
```

```
logInfo("Started block pushing thread")

def areBlocksBeingGenerated: Boolean = synchronized {
  state != StoppedGeneratingBlocks
}

try {
  // While blocks are being generated, keep polling for to-be-pushed blocks and push them.
  while (areBlocksBeingGenerated) {
    // 每10ms从blocksForPushing中取出一个Block，然后调用pushBlock方法
    Option(blocksForPushing.poll(10, TimeUnit.MILLISECONDS)) match {
      case Some(block) => pushBlock(block)
      case None =>
    }
  }

  // At this point, state is StoppedGeneratingBlock. So drain the queue of to-be-pushed blocks.
  logInfo("Pushing out the last "+ blocksForPushing.size() + "blocks")
  // 如果blocksForPushing不为空，则把其中的Block取出，并调用pushBlock方法
  while (!blocksForPushing.isEmpty) {
    val block = blocksForPushing.take()
    logDebug(s"Pushing block $block")
    pushBlock(block)
    logInfo("Blocks left to push " + blocksForPushing.size())
  }
  logInfo("Stopped block pushing thread")
```

```
  } catch {
    case ie: InterruptedException =>
      logInfo("Block pushing thread was interrupted")
    case e: Exception =>
      reportError("Error in block pushing thread", e)
  }
}
```

此线程以10ms为周期不断地从blocksForPushing队列中取Block，并调用pushBlock方法。

源码3-40 BlockGenerator.pushBlock

```
private def pushBlock(block: Block) {
  listener.onPushBlock(block.id, block.buffer)
  logInfo("Pushed block " + block.id)
}
```

listener.onPushBlock中的listener是ReceiverSupervisorImpl.defaultBlockGeneratorListener。

源码3-41 ReceiverSupervisorImpl.defaultBlockGeneratorListener

```
/** Divides received data records into data blocks for pushing in BlockManager. */
private val defaultBlockGeneratorListener = new BlockGeneratorListener {
  def onAddData(data: Any, metadata: Any): Unit = { }

  def onGenerateBlock(blockId: StreamBlockId): Unit = { }
```

```
  def onError(message: String, throwable: Throwable) {
    reportError(message, throwable)
  }

  def onPushBlock(blockId: StreamBlockId, arrayBuffer: ArrayBuffer[_]) {
    pushArrayBuffer(arrayBuffer, None, Some(blockId))
  }
}
```

其中定义了onPushBlock，它调用了pushArrayBuffer。

源码3-42 ReceiverSupervisorImpl.pushArrayBuffer

```
/** Store an ArrayBuffer of received data as a data block into Spark's memory. */
def pushArrayBuffer(
    arrayBuffer: ArrayBuffer[_],
    metadataOption: Option[Any],
    blockIdOption: Option[StreamBlockId]
  ) {
  pushAndReportBlock(ArrayBufferBlock(arrayBuffer), metadataOption, blockIdOption)
}
```

源码3-43 ReceiverSupervisorImpl.pushAndReportBlock

```
/** Store block and report it to driver */
def pushAndReportBlock(
    receivedBlock: ReceivedBlock,
```

```
    metadataOption: Option[Any],
    blockIdOption: Option[StreamBlockId]
  ) {
  val blockId = blockIdOption.getOrElse(nextBlockId)
  val time = System.currentTimeMillis
  val blockStoreResult = receivedBlockHandler.storeBlock(blockId, receivedBlock)
  logDebug(s"Pushed block $blockId in ${(System.currentTimeMillis - time)} ms")
  val numRecords = blockStoreResult.numRecords
  val blockInfo = ReceivedBlockInfo(streamId, numRecords, metadataOption, blockStoreResult)
  trackerEndpoint.askWithRetry[Boolean](AddBlock(blockInfo))
  logDebug(s"Reported block $blockId")
}
```

存储Block并发消息给Driver。存储Block是通过调用receivedBlockHandler.storeBlock实现的。receivedBlockHandler是ReceivedBlockHandler的子类对象。

源码3-44　ReceiverSupervisorImpl.receivedBlockHandler

```
private val receivedBlockHandler: ReceivedBlockHandler = {
  if (WriteAheadLogUtils.enableReceiverLog(env.conf)) {
    if (checkpointDirOption.isEmpty) {
      throw new SparkException(
        "Cannot enable receiver write-ahead log without checkpoint directory set. "+
          "Please use streamingContext.checkpoint() to set the checkpoint directory. "+
          "See documentation for more details.")
    }
```

```
    new WriteAheadLogBasedBlockHandler(env.blockManager, receiver.streamId,
      receiver.storageLevel, env.conf, hadoopConf, checkpointDirOption.get)
  } else {
    new BlockManagerBasedBlockHandler(env.blockManager, receiver.storageLevel)
  }
}
```

可见receivedBlockHandler既可能是WriteAheadLogBasedBlockHandler对象，也可能是BlockManagerBasedBlockHandler对象。这由应用程序运行时的配置来决定。所以receivedBlockHandler.storeBlock既可能是WriteAheadLogBasedBlockHandler.storeBlock，也可能是BlockManagerBasedBlockHandler.StoreBlock。这里以WriteAheadLogBasedBlockHandler为例进行解析。

源码3-45 WriteAheadLogBasedBlockHandler.storeBlock

```
/**
 * This implementation stores the block into the block manager as well as a write ahead log.
 * It does this in parallel, using Scala Futures, and returns only after the block has
 * been stored in both places.
 */
def storeBlock(blockId: StreamBlockId, block: ReceivedBlock): ReceivedBlockStoreResult = {

  var numRecords = None: Option[Long]
  // Serialize the block so that it can be inserted into both
  val serializedBlock = block match {
    case ArrayBufferBlockarrayBuffer() =>
      numRecords = Some(arrayBuffer.size.toLong)
```

```
    blockManager.dataSerialize(blockId, arrayBuffer.iterator)
  case IteratorBlock(iterator) =>
    val countIterator = new CountingIterator(iterator)
    val serializedBlock = blockManager.dataSerialize(blockId, countIterator)
    numRecords = countIterator.count
    serializedBlock
  case ByteBufferBlock(byteBuffer) =>
    byteBuffer
  case _ =>
    throw new Exception(s"Could not push $blockId to block manager, unexpected block type")
}

// Store the block in block manager
val storeInBlockManagerFuture = Future {
  val putResult =
    blockManager.putBytes(blockId, serializedBlock, effectiveStorageLevel, tellMaster = true)
  if (!putResult.map { _._1 }.contains(blockId)) {
    throw new SparkException(
      s"Could not store $blockId to block manager with storage level $storageLevel")
  }
}

// Store the block in write ahead log
val storeInWriteAheadLogFuture = Future {
  writeAheadLog.write(serializedBlock, clock.getTimeMillis())
}

// Combine the futures, wait for both to complete, and return the write ahead log record handle
```

```
    val combinedFuture = storeInBlockManagerFuture.zip(storeInWriteAheadLogFuture).map(_._2)
    val walRecordHandle = Await.result(combinedFuture, blockStoreTimeout)
    WriteAheadLogBasedStoreResult(blockId, numRecords, walRecordHandle)
  }
```

WriteAheadLogBasedBlockHandler.storeBlock会调用blockManager.putBytes。这个blockManager是用Spark Core中的BlockManager生成的对象。

源码3-46 BlockManager.putBytes

```
/**
 * Put a new block of serialized bytes to the block manager.
 * Return a list of blocks updated as a result of this put.
 */
def putBytes(
    blockId: BlockId,
    bytes: ByteBuffer,
    level: StorageLevel,
    tellMaster: Boolean = true,
    effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId, BlockStatus)] = {
  require(bytes != null, "Bytes is null")
  doPut(blockId, ByteBufferValues(bytes), level, tellMaster, effectiveStorageLevel)
}
```

putBytes调用了doPut。BlockManager.doPut主要按照存储级别对Block进行存储，必要时要做复制。源码较长，且是Spark Core的内容，不做重点剖析。

回过头来再看ReceiverSupervisorImpl.pushAndReportBlock（源码3-43），分析到其中的trackerEndpoint.askWithRetry。这个trackerEndpoint是RpcEndpointRef子类的对象。这里发送了一个AddBlock消息。Driver端ReceiverTracker的ReceiverEndpoint成员endpoint会接收到这个消息。

源码3-47　ReceiverTracker.ReceiverEndpoint.ReceiverAndReply

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
 // Remote messages
 case RegisterReceiver(streamId, typ, host, executorId, receiverEndpoint) =>
   ...
 case AddBlock(receivedBlockInfo) =>
   if (WriteAheadLogUtils.isBatchingEnabled(ssc.conf, isDriver = true)) {
     walBatchingThreadPool.execute(new Runnable {
       override def run(): Unit = Utils.tryLogNonFatalError {
         if (active) {
           context.reply(addBlock(receivedBlockInfo))
         } else {
           throw new IllegalStateException("ReceiverTracker RpcEndpoint shut down.")
         }
       }
     })
   } else {
     context.reply(addBlock(receivedBlockInfo))
   }
 case DeregisterReceiver(streamId, message, error) =>
   ...
```

```
 case AllReceiverIds =>
   ...
 case StopAllReceivers =>
   ...
}
```

ReceiverTracker.ReceiverEndpoint.ReceiverAndReply在处理AddBlock消息时调用了addBlock。

源码3-48 ReceiverTracker.addBlock

```
/** Add new blocks for the given stream */
private def addBlock(receivedBlockInfo: ReceivedBlockInfo): Boolean = {
  receivedBlockTracker.addBlock(receivedBlockInfo)
}
```

ReceiverTracker.addBlock调用了ReceivedBlockTracker.addBlock。ReceivedBlockTracker就是用于记录跟踪Receiver发送来的Block的源数据信息。

源码3-49 ReceivedBlockTracker.addBlock

```
/** Add received block. This event will get written to the write ahead log (if enabled). */
def addBlock(receivedBlockInfo: ReceivedBlockInfo): Boolean = {
  try {
    val writeResult = writeToLog(BlockAdditionEvent(receivedBlockInfo))
    if (writeResult) {
      synchronized {
        getReceivedBlockQueue(receivedBlockInfo.streamId) += receivedBlockInfo
```

```
      }
      logDebug(s"Stream ${receivedBlockInfo.streamId} received " +
        s"block ${receivedBlockInfo.blockStoreResult.blockId}")
    } else {
      logDebug(s"Failed to acknowledge stream ${receivedBlockInfo.streamId} receiving "+
        s"block ${receivedBlockInfo.blockStoreResult.blockId} in the Write Ahead Log.")
    }
    writeResult
  } catch {
    case NonFatal(e) =>
      logError(s"Error adding block $receivedBlockInfo", e)
      false
  }
}
```

其中的writeToLog用于更新WAL（预写日志）信息，在3.5节介绍容错机制时再进一步剖析。最关键的代码是把Block信息加入到了指定的Block信息队列中。Block信息队列是以键值对（键是Block信息的streamId，值是Block信息队列）形式存在于HashMap[Int, ReceivedBlockQueue]类型的集合streamIdToUnallocatedBlockQueues中，这就为后续的数据处理时为Job分配接收到的相应数据做好了准备。

3.3 数据处理

Spark Streaming的数据处理就是周期性地不断产生Job并在Executor上去执行处理。

Job是通过JobGenerator生成的。这里说的Job和Spark Core中的Job不是一回事。Spark Streaming中的Job相当于Java中线程要处理的Runnable的业务逻辑的封装，而Spark Core上的Job是一个运行的作业。

JobGenerator会利用DStream来生成Job。

DStream有两类：一类是根据数据源产生DStream，另一类是通过前面的DStream转换后生成DStream。

所以说，JobGenerator是根据DStream的依赖关系，或者说根据DStreamGraph来产生Job。

当然从时间维度上讲，JobGenarator会不断地产生Job。在Spark问世之前，我们做连续不断的大数据任务时，如果不采用流式处理，一般会用定时任务。其实这也是变相地在做流处理。不管是定时1分钟、1小时甚至是1天，都类似在做流处理。原来的定时任务实际上是在做变相的流处理。

先提供Job生成的总体轮廓，如图3-10所示。

ForeachDStream.getOrCompute的流程会随着Spark Streaming应用程序的业务逻辑的不同而不同，后面会结合具体应用程序案例进行剖析。

先看JobGenerator.start。

源码3-50 JobGenerator.start

```
/** Start generation of jobs */
def start(): Unit = synchronized {
  ...
  eventLoop = new EventLoop[JobGeneratorEvent]("JobGenerator") {
    override protected def onReceive(event: JobGeneratorEvent): Unit = processEvent(event)
    ...
    }
  }
  eventLoop.start()
```

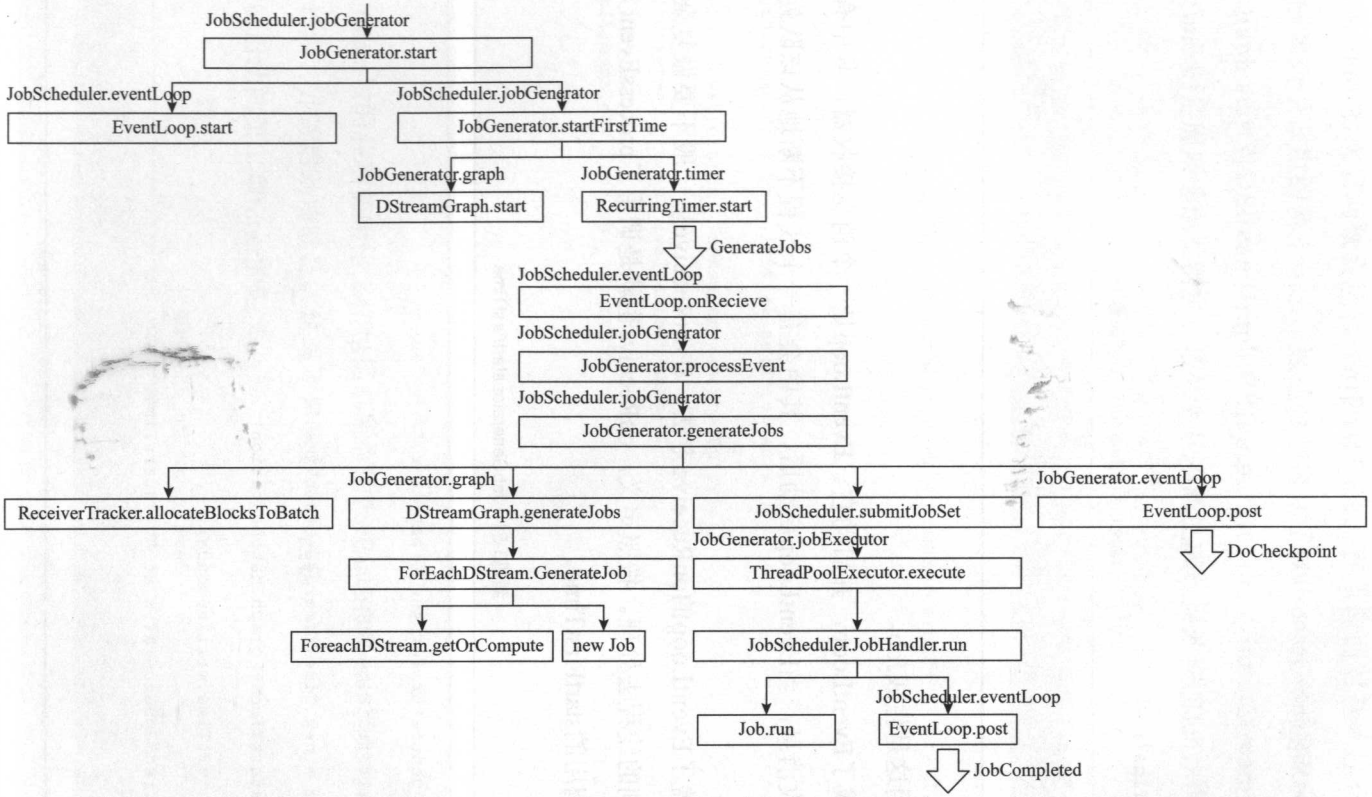


图3-10　JobGenerator正常启动的调用流程

```
  if (ssc.isCheckpointPresent) {
    restart()
  } else {
    startFirstTime()
  }
}
```

注释也说明这是生成Job。

代码中生成了EventLoop，并启动它。EventLoop是一个消息接收器，其中有一个消息队列，用于接收消息，当EventLoop启动后，其中会有一个线程不断地从该队列中取消息进行处理。

代码中覆盖了EventLoop的onReceive方法，在onReceive里一般不要做复杂耗时的操作，应该交给别的线程去处理。按照定义，onReceive也就是调用了processEvent方法。

接下来看调用的startFirstTime。

源码3-51 JobGenerator.startFirstTime

```
/** Starts the generator for the first time */
private def startFirstTime() {
  val startTime = new Time(timer.getStartTime())
  graph.start(startTime - graph.batchDuration)
  timer.start(startTime.milliseconds)
  logInfo("Started JobGenerator at " + startTime)
}
```

JobGenerator初次启动就会运行startFirstTime。有了前面对基本原理的阐述，可以很容易理解源码中调用的两个start，这是分别启动了DStreamGraph和RecurringTimer。DStreamGraph.start对DStreamGraph中的outputStreams、inputStreams做初始设置；RecurringTimer则是周期性地发消息，通知系统不断生成数据处理的Job并执行。

源码3-52 DStreamGraph.start

```
def start(time: Time) {
  this.synchronized {
    require(zeroTime == null, "DStream graph computation already started")
    zeroTime = time
    startTime = time
    outputStreams.foreach(_.initialize(zeroTime))
    outputStreams.foreach(_.remember(rememberDuration))
    outputStreams.foreach(_.validateAtStart)
    inputStreams.par.foreach(_.start())
  }
}
```

对DStream做的各项初始化工作包括：所有的ForEaachDStream初始化，设置计算相关的时间段长度，并做启动前验证。这里不做重点一一剖析。

来看看启动的第二项timer的定义。

源码3-53 JobGenerator.timer

```
private val timer = new RecurringTimer(clock, ssc.graph.batchDuration.milliseconds,
  longTime => eventLoop.post(GenerateJobs(new Time(longTime))), "JobGenerator")
```

第三个参数是匿名的回调函数，其中定义的是发送GenerateJobs消息。在timer启动后，会启动一个线程。数据处理这项周期性的工作是靠timer的这个线程周期性地发送GenerateJobs消息来触发的。

源码3-50中已定义了eventLoop的onReceive就是调用processEvent。

源码3-54 JobGenerator.processEvent

```
/** Processes all events */
private def processEvent(event: JobGeneratorEvent) {
  logDebug("Got event " + event)
  event match {
    case GenerateJobs(time) => generateJobs(time)
    case ClearMetadata(time) => clearMetadata(time)
    case DoCheckpoint(time, clearCheckpointDataLater) =>
      doCheckpoint(time, clearCheckpointDataLater)
    case ClearCheckpointData(time) => clearCheckpointData(time)
  }
}
```

processEvent针对GenerateJobs消息，调用了generateJobs。

源码3-55 JobGenerator.genetateJobs

```
/** Generate jobs and perform checkpoint for the given `time`.  */
     private def generateJobs(time: Time) {
  // Set the SparkEnv in this thread, so that job generation code can access the environment
  // Example: BlockRDDs are created in this thread, and it needs to access BlockManager
```

```
  // Update: This is probably redundant after threadlocal stuff in SparkEnv has been removed.
  SparkEnv.set(ssc.env)
  Try {
    jobScheduler.receiverTracker.allocateBlocksToBatch(time)
     // allocate received blocks to batch
    graph.generateJobs(time) // generate jobs using allocated block
  } match {
    case Success(jobs) =>
      val streamIdToInputInfos = jobScheduler.inputInfoTracker.getInfo(time)
      jobScheduler.submitJobSet(JobSet(time, jobs, streamIdToInputInfos))
    case Failure(e) =>
      jobScheduler.reportError("Error generating jobs for time "+ time, e)
  }
  eventLoop.post(DoCheckpoint(time, clearCheckpointDataLater = false))
}
```

在generateJobs中做了多项工作。首先，前面数据接收部分生成的Block终于通过ReceiverTracker把Block分配给了相应批次的批处理工作，这是由ReceiverTracker.allocateBlocksToBatch完成的。其次，根据应用程序中DStream的依赖关系生成Job，这些由DStreamGraph.generateJobs完成。Job被提交给 Executor执行，这些由JobScheduler.submitJobs完成。最后，为了容错，设置检查点，这个由EventLoop.post发送DoCheckpoint消息来完成。下面做逐个剖析。

源码3-56 ReceiverTracker.allocateBlocksToBatch

```
/** Allocate all unallocated blocks to the given batch. */
```

```
def allocateBlocksToBatch(batchTime: Time): Unit = {
  if (receiverInputStreams.nonEmpty) {
    receivedBlockTracker.allocateBlocksToBatch(batchTime)
  }
}
```

如果ReceiverInputDStream存在，则ReceiverTracker.allocateBlocksToBatch做的实际上就是ReceivedBlockTracker.allocateBlocksToBatch。

源码3-57 ReceivedBlockTracker.allocateBlocksToBatch

```
/**
 * Allocate all unallocated blocks to the given batch.
 * This event will get written to the write ahead log (if enabled).
 */
def allocateBlocksToBatch(batchTime: Time): Unit = synchronized {
  if (lastAllocatedBatchTime == null || batchTime > lastAllocatedBatchTime) {
    val streamIdToBlocks = streamIds.map { streamId =>
        (streamId, getReceivedBlockQueue(streamId).dequeueAll(x => true))
    }.toMap
    val allocatedBlocks = AllocatedBlocks(streamIdToBlocks)
    if (writeToLog(BatchAllocationEvent(batchTime, allocatedBlocks))) {
      timeToAllocatedBlocks.put(batchTime, allocatedBlocks)
      lastAllocatedBatchTime = batchTime
    } else {
```

```
      logInfo(s"Possibly processed batch $batchTime need to be processed again in WAL recovery")
    }
  } else {
    // This situation occurs when:
    // 1. WAL is ended with BatchAllocationEvent, but without BatchCleanupEvent,
    // possibly processed batch job or half-processed batch job need to be processed again,
    // so the batchTime will be equal to lastAllocatedBatchTime.
    // 2. Slow checkpointing makes recovered batch time older than WAL recovered
    // lastAllocatedBatchTime.
    // This situation will only occurs in recovery time.
    logInfo(s"Possibly processed batch $batchTime need to be processed again in WAL recovery")
  }
}
```

把未分配的Block分配给指定的批处理。如果有WAL（Write Ahead Log）设置，则要做好预写日志。

再回去看graph.generateJobs（源码3-55）。给指定批次分配Block后，就是执行代码graph.generateJobs。每个Spark Streaming应用程序因为业务逻辑不一定相同，生成Job的流程也不一定相同。所以这里先给出一个应用程序，针对个案进行分析，以利于理解。仍以NetworkWordCount（源码3-1）为例进行剖析。

先给出NetworkWordCount中DStreamGraph.generateJobs的主要调用流程图，如图3-11所示。

再给出NetworkWordCount中DStreamGraph.generateJobs的UML序列图，如图3-12所示。

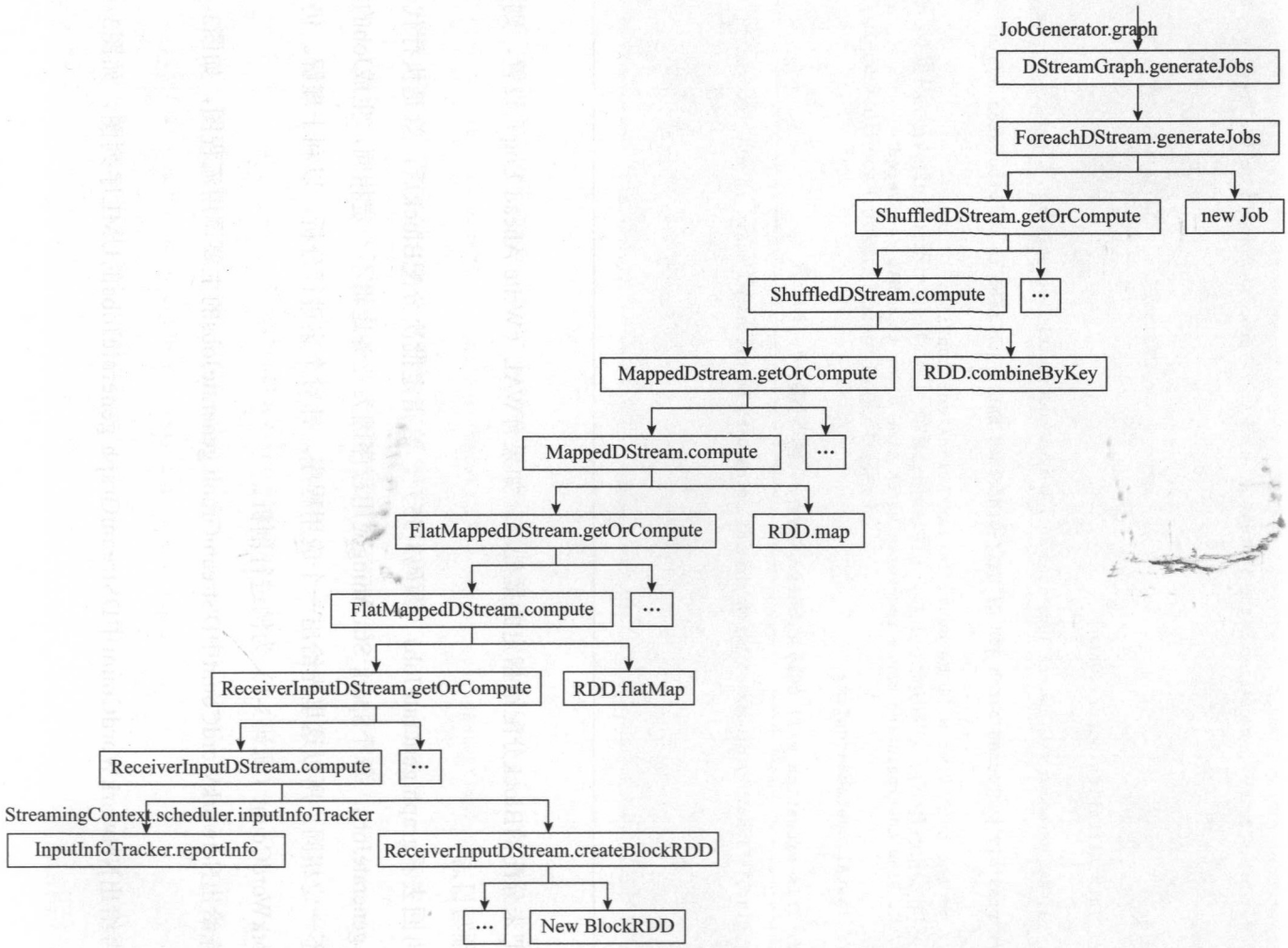


图3-11　NetworkWordCount中DStreamGraph.generateJobs的调用流程

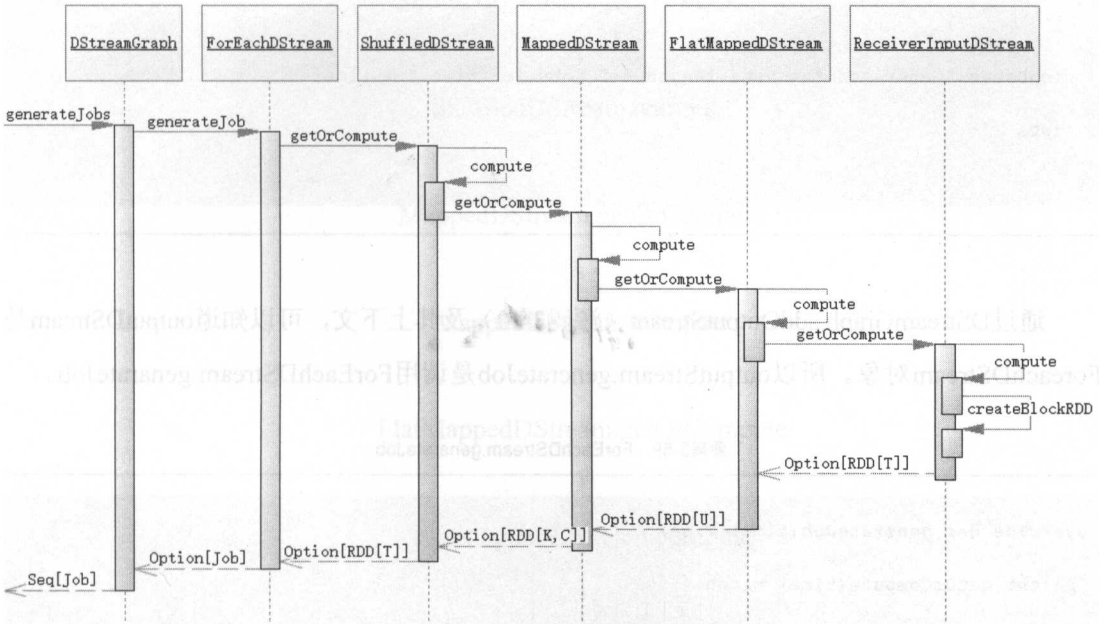


图3-12　NetworkWordCount中DStreamGraph.generateJobs的UML序列图

UML序列图中体现了主要方法的返回值类型，省略了部分类和方法，以方便分析各DStream的流程。

下面从DStreamGraph.generateJobs开始剖析源码。

源码3-58　DStreamGraph.generateJobs

```
def generateJobs(time: Time): Seq[Job] = {
  logDebug("Generating jobs for time " + time)
  val jobs = this.synchronized {
    outputStreams.flatMap { outputStream =>
      val jobOption = outputStream.generateJob(time)
      jobOption.foreach(_.setCallSite(outputStream.creationSite))
      jobOption
    }
```

```
  }
  logDebug("Generated " + jobs.length + " jobs for time " + time)
  jobs
}
```

通过DStreamGraph.addOutputStream（源码3-10）及其上下文，可以知道outputDStream是ForeachDStream对象。所以outputStream.generateJob是调用ForEachDStream.genarateJob。

源码3-59 ForEachDStream.genarateJob

```
override def generateJob(time: Time): Option[Job] = {
 parent.getOrCompute(time) match {
   case Some(rdd) =>
     val jobFunc = () => createRDDWithLocalProperties(time, displayInnerRDDOps) {
       foreachFunc(rdd, time)
     }
     Some(new Job(time, jobFunc))
   case None => None
 }
}
```

事实上，genarateJob中通过父类DStream的getOrCompute与例程中各个DStream子类的compute方法组成了职责链模式。这个调用顺序如下：

ForEachDStream.generateJob

↓

ShuffledDStream.getOrCompute

↓

ShuffledDStream.compute

↓

MappedDStream.getOrCompute

↓

MappedDStream.compute

FlatMappedDStream.getOrCompute

↓

FlatMappedDStream.compute

↓

ReceiverInputDStream.getOrCompute

↓

ReceiverInputDStream.compute

DStream各子类没有覆写getOrCompute，所以实际是调用DStream.getOrCompute。

源码3-60　DStream.getOrCompute

```
private[streaming] final def getOrCompute(time: Time): Option[RDD[T]] = {
  generatedRDDs.get(time).orElse {
    if (isTimeValid(time)) {
      ...
        PairRDDFunctions.disableOutputSpecValidation.withValue(true) {
        compute(time)
      }
    }
```

```
      ...
    } else {
      None
    }
  }
}
```

源码3-61 ShuffledDStream.compute

```
override def compute(validTime: Time): Option[RDD[(K, C)]] = {
  parent.getOrCompute(validTime) match {
    case Some(rdd) => Some(rdd.combineByKey[C](
        createCombiner, mergeValue, mergeCombiner, partitioner, mapSideCombine))
    case None => None
  }
}
```

源码3-62 MappedDStream.compute

```
override def compute(validTime: Time): Option[RDD[U]] = {
  parent.getOrCompute(validTime).map(_.map[U](mapFunc))
}
```

源码3-63 FlatMappedDStream.compute

```
override def compute(validTime: Time): Option[RDD[U]] = {
```

```
  parent.getOrCompute(validTime).map(_.flatMap(flatMapFunc))
}
```

源码3-64　ReceiverInputDStream.compute

```
/**
 * Generates RDDs with blocks received by the receiver of this stream. */
override def compute(validTime: Time): Option[RDD[T]] = {
  val blockRDD = {

    if (validTime < graph.startTime) {
      // If this is called for any time before the start time of the context,
      // then this returns an empty RDD. This may happen when recovering from a
      // driver failure without any write ahead log to recover pre-failure data.
      new BlockRDD[T](ssc.sc, Array.empty)
    } else {
      // Otherwise, ask the tracker for all the blocks that have been allocated to this stream
      // for this batch
      val receiverTracker = ssc.scheduler.receiverTracker
      val blockInfos = receiverTracker.getBlocksOfBatch(validTime).getOrElse(id, Seq.empty)

      // Register the input blocks information into InputInfoTracker
      val inputInfo = StreamInputInfo(id, blockInfos.flatMap(_.numRecords).sum)
      ssc.scheduler.inputInfoTracker.reportInfo(validTime, inputInfo)

    // Create the BlockRDD
```

```
      createBlockRDD(validTime, blockInfos)
    }
  }
  Some(blockRDD)
}
```

以上的DStream子类的compute方法都是返回新的RDD对象。

注意，这些DStream子类中的parent是指被依赖的DStream。

只有职责链最后的ReceiverInputDStream因为是接收源数据的DStream，不依赖其他DStream，所以必须自己生成RDD。生成RDD时，会从ReceiverTracker那里获取待分配的数据块信息，接着调用createBlockRDD。

源码3-65 ReceiverInputDStream.createBlockRDD

```
private[streaming] def createBlockRDD(time: Time, blockInfos: Seq[ReceivedBlockInfo]): RDD[T] = {

    if (blockInfos.nonEmpty) {
      val blockIds = blockInfos.map { _.blockId.asInstanceOf[BlockId] }.toArray

      // Are WAL record handles present with all the blocks
      val areWALRecordHandlesPresent = blockInfos.forall { _.walRecordHandleOption.nonEmpty }

      if (areWALRecordHandlesPresent) {
        // If all the blocks have WAL record handle, then create a WALBackedBlockRDD
        val isBlockIdValid = blockInfos.map { _.isBlockIdValid() }.toArray
        val walRecordHandles = blockInfos.map { _.walRecordHandleOption.get }.toArray
```

```
      new WriteAheadLogBackedBlockRDD[T](
        ssc.sparkContext, blockIds, walRecordHandles, isBlockIdValid)
    } else {
      // Else, create a BlockRDD. However, if there are some blocks with WAL info but not
      // others then that is unexpected and log a warning accordingly.
      if (blockInfos.find(_.walRecordHandleOption.nonEmpty).nonEmpty) {
        if (WriteAheadLogUtils.enableReceiverLog(ssc.conf)) {
          logError("Some blocks do not have Write Ahead Log information; "+
            "this is unexpected and data may not be recoverable after driver failures")
        } else {
          logWarning("Some blocks have Write Ahead Log information; this is unexpected")
        }
      }
      val validBlockIds = blockIds.filter { id =>
        ssc.sparkContext.env.blockManager.master.contains(id)
      }
      if (validBlockIds.size != blockIds.size) {
        logWarning("Some blocks could not be recovered as they were not found in memory. "+
          "To prevent such data loss, enabled Write Ahead Log (see programming guide "+
          "for more details.")
      }
      new BlockRDD[T](ssc.sc, validBlockIds)
    }
  } else {
    // If no block is ready now, creating WriteAheadLogBackedBlockRDD or BlockRDD
    // according to the configuration
```

```
      if (WriteAheadLogUtils.enableReceiverLog(ssc.conf)) {
        new WriteAheadLogBackedBlockRDD[T](
          ssc.sparkContext, Array.empty, Array.empty, Array.empty)
      } else {
        new BlockRDD[T](ssc.sc, Array.empty)
      }
    }
  }
```

生成了RDD，分配了相应的Block数据。

在ReceiverInputDStream.compute后沿责任链返回时，在各DStream子类（FlatMappedStream、MappedDStream、ShuffleDStream）的compute中的getOrCompute后，还会有各自的RDD转换操作。RDD的转换操作顺序是RDD.flatMap → RDD.map → RDD.combineByKey。

这些RDD转换操作和其他Spark应用程序中的一样，不会马上执行，要等后续有RDD的action操作出现。

最后职责链又回到ForEachDStream.generateJob。

源码3-66 ForEachDStream.generateJob

```
override def generateJob(time: Time): Option[Job] = {
 parent.getOrCompute(time) match {
   case Some(rdd) =>
     val jobFunc = () => createRDDWithLocalProperties(time, displayInnerRDDOps) {
       foreachFunc(rdd, time)
     }
```

```
      Some(new Job(time, jobFunc))
    case None => None
  }
}
```

这段代码实际和源码3-59是同一段代码。只是强调的重点有不同。

在parent.getOrCompute返回的是RDD对象时定义函数jobFunc。

注意，jobFunc函数在这里只是被定义，而不是被调用。jobFunc函数中会调用foreachFunc函数。但是jobFunc函数没有被调用，所以foreachFunc函数也并没有实际执行。

RDD子类对象会随jobFunc函数封装在新生成的Job中。

每个ForEachDStream都会生成一个Job。

Job就这样生成了，接下来要看如何分发业务功能到Executor上去执行。

回过头来看JobGenerator.generateJobs（源码3-55）中的jobScheduler.submitJobSet。

源码3-67　JobScheduler.submitJobSet

```
def submitJobSet(jobSet: JobSet) {
  if (jobSet.jobs.isEmpty) {
    logInfo("No jobs added for time " + jobSet.time)
  } else {
    listenerBus.post(StreamingListenerBatchSubmitted(jobSet.toBatchInfo))
    jobSets.put(jobSet.time, jobSet)
    jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))
    logInfo("Added jobs for time " + jobSet.time)
  }
}
```

jobExecutor是线程池，JobHandler是Runnable的子类。JobExecutor.execute在线程池中运行JobHandler所对应的Job。

源码3-68 JobHandler

```
private class JobHandler(job: Job) extends Runnable with Logging {
  import JobScheduler._
   def run() {
    try {
      ...
      if (_eventLoop != null) {
        ...
        PairRDDFunctions.disableOutputSpecValidation.withValue(true) {
          job.run()
        }
        ...
      } else {
        // JobScheduler has been stopped.
      }
    } finally {
      ssc.sc.setLocalProperty(JobScheduler.BATCH_TIME_PROPERTY_KEY, null)
      ssc.sc.setLocalProperty(JobScheduler.OUTPUT_OP_ID_PROPERTY_KEY, null)
    }
  }
}
```

JobHandler.run会调用job.run。

源码3-69 Job片段

```
class Job(val time: Time, func: () => _) {
  private var _id: String = _
  private var _outputOpId: Int = _
  private var isSet = false
  private var _result: Try[_] = null
  private var _callSite: CallSite = null
  private var _startTime: Option[Long] = None
  private var _endTime: Option[Long] = None

  def run() {
    _result = Try(func())
  }
  ...
}
```

此处的Job.run执行了Job自己的函数成员func。

接下来就看Job中的这个函数func是运行些什么。

从3.1节的DStream输出操作分析中，可以知道DStream输出操作中定义的函数，就是ForEachDStream的这个函数成员。

而DStreamGraph.generateJobs（源码3-58）在生成Job时就是利用graph的outputStreams中的每个ForEachDStream调用ForEachDStream.generateJob（源码 3-66）。ForEachDStream的这个函数成员又封装在Job的函数成员中。

所以，DStream输出操作中定义的函数按照以下路径传递：DStream输出操作中定义的函数 → ForEachDStream中的函数成员 → Job的函数成员。

Job.run中执行的函数实际上就是DStream输出操作中定义的函数。

接下来继续分析DStream的输出操作中定义的函数。

如果读者学习过Spark Core，就会知道，Spark应用程序在集群环境中提交业务逻辑到Executor上运行，就是因为有RDD的action操作触发。那么源码3-69的这个负责Job运行的Job.run中调用的函数（即DStream的输出操作中定义的函数），是否和一般的Spark程序一样，也调用了RDD的action操作呢？

仍以DStream的输出操作print（源码3-5）为例进行分析。其内部定义的函数中有一个循环，里面调用了RDD的take方法。RDD.take就是RDD的一个action操作。

源码3-70 RDD.take

```
def take(num: Int): Array[T] = withScope {
  if (num == 0) {
    new Array[T](0)
  } else {
    val buf = new ArrayBuffer[T]
    val totalParts = this.partitions.length
    var partsScanned = 0
    while (buf.size < num && partsScanned < totalParts) {
      // The number of partitions to try in this iteration. It is ok for this number to be
      // greater than totalParts because we actually cap it at totalParts in runJob.
      var numPartsToTry = 1
      if (partsScanned > 0) {
        // If we didn't find any rows after the previous iteration, quadruple and retry.
        // Otherwise, interpolate the number of partitions we need to try, but overestimate
        // it by 50%. We also cap the estimation in the end.
        if (buf.size == 0) {
```

```
          numPartsToTry = partsScanned * 4
        } else {
          // the left side of max is >=1 whenever partsScanned >= 2
          numPartsToTry = Math.max((1.5 * num * partsScanned / buf.size).toInt - partsScanned, 1)
          numPartsToTry = Math.min(numPartsToTry, partsScanned * 4)
        }
      }

      val left = num - buf.size
      val p = partsScanned until math.min(partsScanned + numPartsToTry, totalParts)
      val res = sc.runJob(this, (it: Iterator[T]) => it.take(left).toArray, p)

      res.foreach(buf ++= _.take(num - buf.size))
      partsScanned += numPartsToTry
    }

    buf.toArray
  }
}
```

RDD的take操作是action操作，会循环调用SparkContext.runJob方法提交任务。这属于Spark Core的内容，不做深入分析。

当然，不同的Spark Streaming应用程序的输出操作会有所不同，不一定是本次分析的DStream.print，但一定会调用RDD的某个action操作，继而调用SparkContext.runJob，最终执行提交的作业。

至此，通过Job的运行，数据处理工作被分发到Executor中去执行。

这里揭示一些源码流程之外的DStream操作的技术内幕。

刚才解释了DStream的输出操作如何通过使用定义的函数实现数据处理工作的派发。而实际上，可以借鉴DStream输出操作的定义方式，对DStream（或其子类）定制自己的方法，只要使函数定义中不含有RDD.take这样的action操作，也就不会产生Job，不会调用SparkContxt.runJob。使用自定义函数和foreachRDD操作的方式为开发者提供了后门，可以根据业务需要操作RDD，甚至可以在应用程序中直接写DStream子类.foreachRDD，写一个匿名函数作为参数就可以了。DStream.foreachRDD可以用来编写外部访问功能，比如操作数据库等。

既然说了DStream的输出操作foreachRDD可以不触发Job的生成和执行，就不得不提一下，DStream的转换操作transform却有可能触发Job的生成和执行。

源码3-71 DStream.transform

```
/**
 * Return a new DStream in which each RDD is generated by applying a function
 * on each RDD of 'this' DStream.
 */
def transform[U: ClassTag](transformFunc: (RDD[T], Time) => RDD[U]): DStream[U] = ssc.withScope {
  // because the DStream is reachable from the outer object here, and because
  // DStreams can't be serialized with closures, we can't proactively check
  // it for serializability and so we pass the optional false to SparkContext.clean
  val cleanedF = context.sparkContext.clean(transformFunc, false)
  val realTransformFunc = (rdds: Seq[RDD[_]], time: Time) => {
    assert(rdds.length == 1)
    cleanedF(rdds.head.asInstanceOf[RDD[T]], time)
  }
```

```
  new TransformedDStream[U](Seq(this), realTransformFunc)
}
```

这个transformFunc函数也是对RDD进行操作，然后返回RDD。开发者可以在transformFunc函数中定义RDD的action操作，从而触发Job的生成和执行，只要在action操作之后再保证函数返回的也是RDD就行。DStream.transform可以用来先对某中间计算结果做逻辑判断，再决定后续的相应的RDD操作。

虽然目前已经把Spark Streaming的数据接收、数据处理这两个主要流程进行了深入剖析，但仍有很多其他细节必须深入下去。从下一节开始，将剖析这些细节。

3.4 数据清理

Spark Streaming应用是持续不断地运行着的。如果不对内存资源进行有效管理，内存就有可能很快就耗尽。Spark Streaming应用有自己的对象、数据、元数据的清理机制。

Spark Streaming应用中的对象、数据、元数据是操作DStream时产生的。

先给出数据清理的总流程图，如图3-13所示。

前面分析JobGenerator的启动（可参考图3-10）时分析过JobScheduler.JobHandler.run，当时主要是分析了job.run。其实在正常流程中，job.run的后面部分还会发送一个消息。

源码3-72　JobScheduler.JobHandler.run

```
def run() {
  try {
    ...
```

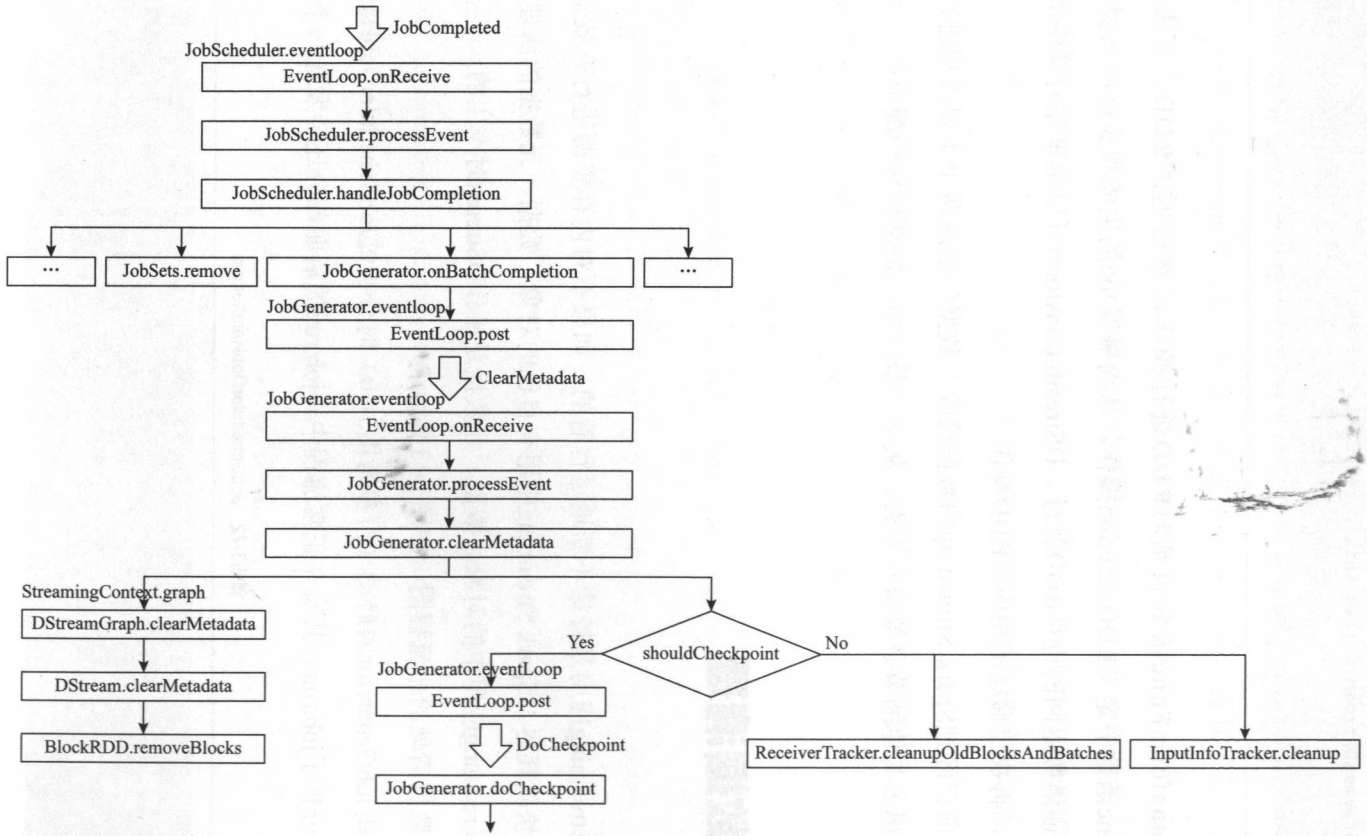


图3-13　数据清理的流程

```
    var _eventLoop = eventLoop
    if (_eventLoop != null) {
      _eventLoop.post(JobStarted(job, clock.getTimeMillis()))
      PairRDDFunctions.disableOutputSpecValidation.withValue(true) {
        job.run()
      }
      _eventLoop = eventLoop
      if (_eventLoop != null) {
        _eventLoop.post(JobCompleted(job, clock.getTimeMillis()))
      }
    } else {
      // JobScheduler has been stopped.
    }
  } finally {
    ssc.sc.setLocalProperty(JobScheduler.BATCH_TIME_PROPERTY_KEY, null)
    ssc.sc.setLocalProperty(JobScheduler.OUTPUT_OP_ID_PROPERTY_KEY, null)
  }
}
```

run发送了JobCompleted 消息。JobScheduler.processEvent定义了针对消息的处理。

源码3-73 JobScheduler.processEvent

```
private def processEvent(event: JobSchedulerEvent) {
  try {
    event match {
```

```
      case JobStarted(job, startTime) => handleJobStart(job, startTime)
      case JobCompleted(job, completedTime) => handleJobCompletion(job, completedTime)
      case ErrorReported(m, e) => handleError(m, e)
    }
  } catch {
    case e: Throwable =>
      reportError("Error in job scheduler", e)
  }
}
```

对JobCompleted事件的处理是调用了handleJobCompletion。

源码3-74 JobScheduler.handleJobCompletion

```
private def handleJobCompletion(job: Job, completedTime: Long) {
  val jobSet = jobSets.get(job.time)
  jobSet.handleJobCompletion(job)
  job.setEndTime(completedTime)
  listenerBus.post(StreamingListenerOutputOperationCompleted(job.toOutputOperationInfo))
  logInfo("Finished job "+ job.id + "from job set of time "+ jobSet.time)
  if (jobSet.hasCompleted) {
    jobSets.remove(jobSet.time)
    jobGenerator.onBatchCompletion(jobSet.time)
    logInfo("Total delay: %.3f s for time %s (execution: %.3f s)".format(
      jobSet.totalDelay / 1000.0, jobSet.time.toString,
      jobSet.processingDelay / 1000.0
```

```
    ))
    listenerBus.post(StreamingListenerBatchCompleted(jobSet.toBatchInfo))
  }
  job.result match {
    case Failure(e) =>
      reportError("Error running job " + job, e)
    case _ =>
  }
}
```

清理JobSets中已提交执行的JobSet，还调用了jobGenerator.onBatchCompletion。

源码3-75　JobGenerator.onBatchCompletion

```
/**
 * Callback called when a batch has been completely processed.
 */
def onBatchCompletion(time: Time) {
  eventLoop.post(ClearMetadata(time))
}
```

发送了ClearMetadata消息。下面看JobGenerator.start中eventLoop的定义。

源码3-76　JobGenerator.start片段

```
eventLoop = new EventLoop[JobGeneratorEvent]("JobGenerator") {
```

```
  override protected def onReceive(event: JobGeneratorEvent): Unit = processEvent(event)

  override protected def onError(e: Throwable): Unit = {
    jobScheduler.reportError("Error in job generator", e)
  }
}
```

由此可知，消息是由eventLoop.onReceive指定的JobGenerator.processEvent做处理。

源码3-77 JobGenerator.processEvent

```
/** Processes all events */
private def processEvent(event: JobGeneratorEvent) {
  logDebug("Got event " + event)
  event match {
    case GenerateJobs(time) => generateJobs(time)
    case ClearMetadata(time) => clearMetadata(time)
    case DoCheckpoint(time, clearCheckpointDataLater) =>
      doCheckpoint(time, clearCheckpointDataLater)
    case ClearCheckpointData(time) => clearCheckpointData(time)
  }
}
```

其中针对清理元数据（ClearMetadata）消息的处理是clearMetadata。

源码3-78　JobGenerator.clearMetadata

```
/** Clear DStream metadata for the given `time`. */
private def clearMetadata(time: Time) {
  ssc.graph.clearMetadata(time)

  // If checkpointing is enabled, then checkpoint,
  // else mark batch to be fully processed
  if (shouldCheckpoint) {
    eventLoop.post(DoCheckpoint(time, clearCheckpointDataLater = true))
  } else {
    // If checkpointing is not enabled, then delete metadata information about
    // received blocks (block data not saved in any case). Otherwise, wait for
    // checkpointing of this batch to complete.
    val maxRememberDuration = graph.getMaxInputStreamRememberDuration()
    jobScheduler.receiverTracker.cleanupOldBlocksAndBatches(time - maxRememberDuration)
    jobScheduler.inputInfoTracker.cleanup(time - maxRememberDuration)
    markBatchFullyProcessed(time)
  }
}
```

可以看到有多项清理工作。注意receiverTracker和inputInfoTracker的清理工作有前提条件：不需要设置检查点。

先看ssc.graph.clearMetadata(time)。

源码3-79 DStreamGraph.clearMetadata

```
def clearMetadata(time: Time) {
  logDebug("Clearing metadata for time " + time)
  this.synchronized {
    outputStreams.foreach(_.clearMetadata(time))
  }
  logDebug("Cleared old metadata for time " + time)
}
```

其中清理了所有OutputDStream的一些相关元数据。看看清除些什么。

源码3-80 DStream.clearMetadata

```
/**
 * Clear metadata that are older than `rememberDuration` of this DStream.
 * This is an internal method that should not be called directly. This default
 * implementation clears the old generated RDDs. Subclasses of DStream may override
 * this to clear their own metadata along with the generated RDDs.
 */
private[streaming] def clearMetadata(time: Time) {
  val unpersistData = ssc.conf.getBoolean("spark.streaming.unpersist", true)
  val oldRDDs = generatedRDDs.filter(_._1 <= (time - rememberDuration))
  logDebug("Clearing references to old RDDs: [" +
    oldRDDs.map(x => s"${x._1} -> ${x._2.id}").mkString(", ") + "]")
```

```
  generatedRDDs --= oldRDDs.keys
  if (unpersistData) {
    logDebug("Unpersisting old RDDs: "+ oldRDDs.values.map(_.id).mkString(", "))
    oldRDDs.values.foreach { rdd =>
      rdd.unpersist(false)
      // Explicitly remove blocks of BlockRDD
      rdd match {
        case b: BlockRDD[_] =>
          logInfo("Removing blocks of RDD " + b + " of time " + time)
          // 将RDD占用的内存块从BlockManager释放
          b.removeBlocks()
        case _ =>
      }
    }
  }
  logDebug("Cleared " + oldRDDs.size + " RDDs that were older than " +
    (time - rememberDuration) + ": " + oldRDDs.keys.mkString(", "))
  dependencies.foreach(_.clearMetadata(time))
}
```

DStream.clearMetadata会清理掉当前DStream的rememberDuration之前的元数据。DStream的子类会覆写此方法。rememberDuration是DStream的成员。

spark.streaming.unpersist（默认是true）的配置可以用来设置是否自动非持久化。这可以显著地减少Spark在RDD上的内存使用，同时也可以改善GC行为。

rememberDuration在启动DStreamGraph时被设置，可参考DStream.initialize，这里不做更多剖析。源码中的设置大体是 slideDuration，如果设置了checkpointDuration则是2*checkpointDuration。还可以通过DStreamGraph.rememberDuration设置：如果想自行设置，可在应用程序中使用StreamingContext.remember方法，不过自行设置的值要大于内部计算得到的值才会生效。

另外值得一提的，就是后面的DStream会调整前面的DStream的rememberDuration，例如，如果用了窗口操作，会有跨若干Batch Duration的情况，则在此之前的DStream的rememberDuration都需要加上windowDuration。不清楚窗口操作的读者可以参考3.10节的相关内容。

最后把依赖的RDD也清理掉了。这是递归调用。

下面分析其中调用的BlockRDD.removeBlocks。

源码3-81 BlockRDD.removeBlocks

```
/**
 * Remove the data blocks that this BlockRDD is made from. NOTE: This is an
 * irreversible operation, as the data in the blocks cannot be recovered back
 * once removed. Use it with caution.
 */
private[spark] def removeBlocks() {
  blockIds.foreach { blockId =>
    sparkContext.env.blockManager.master.removeBlock(blockId)
  }
  _isValid = false
}
```

利用BlockManagerMaster删除当前RDD相关的所有Block。

回过头再看源码3-78。如果需要设置检查点，则发送DoCheckpoint消息。

DoCheckpoint消息的发送留在3.5节关于容错机制的内容中分析。下面分析不需要设置检查点的情况。

先看jobScheduler.receiverTracker.cleanupOldBlocksAndBatches。

源码3-82　ReceiverTracker.cleanupOldBlocksAndBatches

```
/**
 * Clean up the data and metadata of blocks and batches that are strictly
 * older than the threshold time. Note that this does not
 */
def cleanupOldBlocksAndBatches(cleanupThreshTime: Time) {
  // Clean up old block and batch metadata
  receivedBlockTracker.cleanupOldBatches(cleanupThreshTime, waitForCompletion = false)

  // Signal the receivers to delete old block data
  if (WriteAheadLogUtils.enableReceiverLog(ssc.conf)) {
    logInfo(s"Cleanup old received batch data: $cleanupThreshTime")
    synchronized {
      if (isTrackerStarted) {
        endpoint.send(CleanupOldBlocks(cleanupThreshTime))
      }
    }
  }
}
```

ReceiverTracker.cleanupOldBlocksAndBatches调用了ReceivedBlockTracker.cleanup

OldBatches。

源码3-83 ReceivedBlockTracker.cleanupOldBatches

```
/**
 * Clean up block information of old batches. If waitForCompletion is true, this method
 * returns only after the files are cleaned up.
 */
def cleanupOldBatches(cleanupThreshTime: Time, waitForCompletion: Boolean): Unit = synchronized {
  require(cleanupThreshTime.milliseconds < clock.getTimeMillis())
  val timesToCleanup = timeToAllocatedBlocks.keys.filter { _ < cleanupThreshTime }.toSeq
  logInfo("Deleting batches " + timesToCleanup)
  if (writeToLog(BatchCleanupEvent(timesToCleanup))) {
    timeToAllocatedBlocks --= timesToCleanup
    writeAheadLogOption.foreach(_.clean(cleanupThreshTime.milliseconds, waitForCompletion))
  } else {
    logWarning("Failed to acknowledge batch clean up in the Write Ahead Log.")
  }
}
```

cleanupOldBatches清理了旧Batch的Block元数据。

最后回到JobGenerator.clearMetadata（源码3-78），看一看jobScheduler.inputInfoTracker.cleanup。

源码3-84 InputInfoTracker.cleanup

```
/** Cleanup the tracked input information older than threshold batch time */
```

```
def cleanup(batchThreshTime: Time): Unit = synchronized {
  val timesToCleanup = batchTimeToInputInfos.keys.filter(_ < batchThreshTime)
  logInfo(s"remove old batch metadata: ${timesToCleanup.mkString("")}")
  batchTimeToInputInfos --= timesToCleanup
}
```

cleanup清理了旧Batch的跟踪输入源的元数据信息。

3.5 容错机制

Spark Streaming应用程序是周而复始不间断地运行下去的。但是，网络异常、CPU负荷过高、内存溢出、硬盘读写异常、断电等各种内外因素都可能使正在运行着的程序出现异常。

所以Spark Streaming必须有容错机制，以避免或减少故障造成的负面影响，而且能让运行的系统从故障中恢复过来，使流数据处理仍能继续正常进行下去。

Spark Streaming的数据处理实际上也是基于RDD的，所以继承了Spark Core的容错机制，有基于继承关系（lineage）的高度容错机制。计算出错后会从出错位置重新计算，而不会导致从头开始的完全重复计算。

除了RDD相关的计算，Spark Streaming还有数据接收的容错问题以及接收与计算输出衔接等多方面的容错问题。所以除了Spark Core的容错机制，Spark Streaming还会有一些自己独特的容错机制。容错机制是在备份的基础上，在有异常时能利用备份来进行恢复。

先谈谈什么情况下做容错。

如果Spark Streaming应用并不在乎数据的丢失，也就不需要通过备份来容错。例如，

源数据是不断从互联网上爬取的非业务数据，且这种数据以后还有可能被再次爬取到，还有机会被接收并处理，对业务无实质性影响。也就是说，数据可以被丢失，被忽略。值得一提的是，这里所说的忽略是指一个批次（batch）中的所有数据被忽略。Spark Streaming目前还没有做到批次内的部分数据被忽略。

此外，如果上游的源数据系统支持重放，也没必要通过备份来做容错。例如HDFS、Kafka就都支持重放。但考虑好是否要有事务处理的Exactly once要求。仅靠Spark Streaming本身的容错机制并没有解决Exactly once问题。这个问题，需要依赖支持重放的上游源数据系统、No Receiver方式甚至Spark Streaming应用程序的业务处理方式来共同解决，在3.6节中再对此详细分析。

所以，需要使用Spark Streaming的容错机制的情况是：Spark Streaming应用不容许数据未处理就丢失，而且上游源数据系统不支持重放。

3.5.1 容错原理

这里讲的容错实际上就是备份以及利用备份进行恢复。

先讲解容错机制中的冷备、热备这两个备份概念。

- 热备，即热备份。系统在正常工作时，在物理上做好备用硬件运行所需要的状态更新。遇到异常时，系统把受影响的工作自动切换到备用硬件上运行，以保证继续不间断进行。
- 冷备，即冷备份。系统在正常工作时就做好日常数据和元数据的备份。遇到异常时，系统通常需要通过人工重新启动，并利用备份的数据和元数据来恢复工作。

Spark Streaming的热备是针对Executor进行的。其原理基本仍是建立在Spark本身已有的容错机制上。具体剖析在3.5.3节中。这里要讲的容错主要是Spark Streaming的冷备及其恢复。

这里着重介绍Spark Streaming中的冷备及其恢复原理。

在Spark Streaming中，冷备有预写日志（Write Ahead Log，WAL）和冷备检查点（Checkpoint）这两种常用方法。

WAL冷备是备份与后续工作相关的接收数据及其块的元数据信息。

Checkpoint冷备则是为在出错后恢复而做Spark Streaming应用程序配置、DStream操作、未处理作业等元数据信息的备份。有状态操作时也需要设置检查点。

下面简单介绍一下冷备的工作流程。

当一个Spark Streaming应用启动了（例如Driver启动），相应的StreamingContext使用SparkContext去启动Receiver。Receiver是一个长时间执行的作业，这些接收器接收并保存这些数据到Spark的Executor进程的内存中，这些数据的生命周期如图3-14所示。

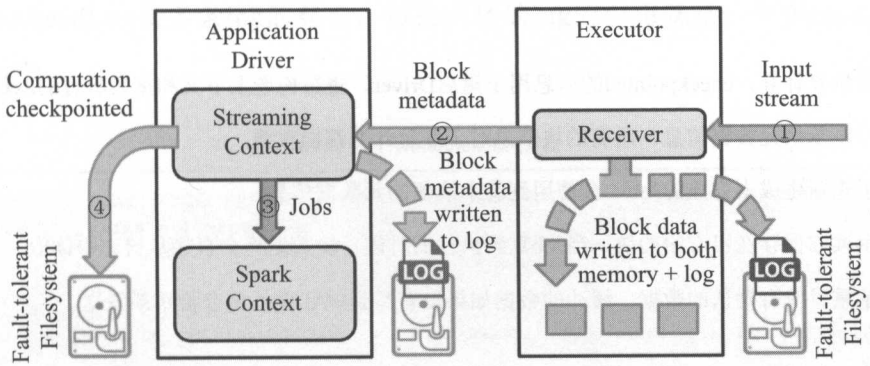


图3-14　冷备的流程

图中：

①表示接收的数据，接收器把数据流打包成块，存储在Executor的内存中，如果开启了WAL，将会把数据写入到存在容错文件系统的日志文件中。

②表示提醒Driver将接收到的数据块的元信息发送给Driver中的StreamingContext，这些元数据包括Executor内存中数据块的引用ID和日志文件中数据块的偏移信息。

③表示处理数据，每一个批处理间隔，StreamingContext使用块信息生成RDD和Job。SparkContext执行这些Job用于处理Executor内存中的数据块。

④表示做这些计算的Checkpoint，以便于恢复。流式处理会周期性地通过检查点设置保存到文件中。

当Driver因失败而重启以后，恢复流程如图3-15所示。

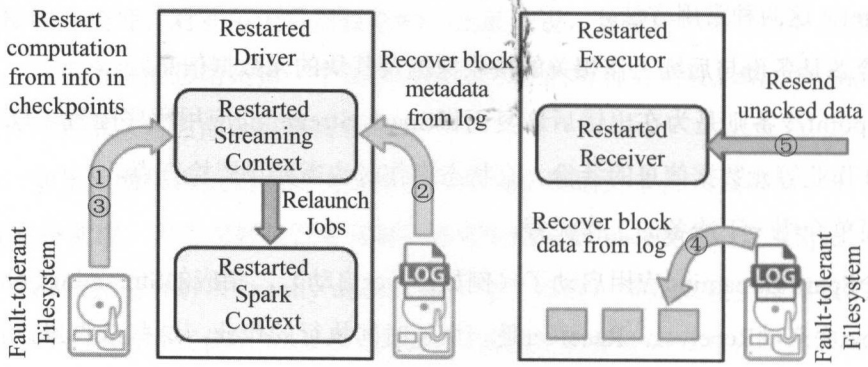


图3-15　用冷备恢复的流程

图中：

①表示恢复计算，checkpointed的信息用于重启Driver，重新构造上下文和重启所有的Receiver。

②表示恢复块元数据信息，所有的块信息对于恢复计算都很重要。

③表示重新生成未完成的Job，会使用到②中恢复的元数据信息。

④表示读取保存在日志中的块，当Job重新执行的时候，块数据将会直接从日志中读取。

⑤表示重发没有确认的数据。缓冲的数据如果没有写到WAL中，将会被重新发送。

下面对WAL、Checkpoint的原理分别做进一步分析。

1. WAL

先介绍一下WAL框架。

WAL 的方式在单机 RDBMS、NoSQL/NewSQL 中都有广泛应用。例如，数据库记录 transaction log，HBase插入数据可以先写到HLog里。

WAL的特点是顺序写入，所以在做数据备份时效率较高，但在需要恢复数据时又需要顺序读取，所以需要一定恢复时间（recovery time）。

不过对于 Spark Streaming 的块数据冷备来讲，在恢复时也非常方便。这是因为对某个块数据的操作只有一次（即新增块数据），而没有后续对块数据的追加、修改、删除操作，这就使得在WAL里只会有一条此块数据的日志入口（log entry）。所以，在恢复时只

要找到这条日志入口并读取就可以了，不需要顺序读取整个WAL。

从效率上看，Spark Streaming基于WAL冷备进行恢复，需要的恢复时间只是找到并读一条日志入口的时间，而不是读取整个WAL的时间，这样非常节省时间。

启动WAL需要做如下的配置：

（1）给streamingContext设置Checkpoint的目录，该目录必须是Hadoop支持的文件系统，用来保存WAL和做Streaming的Checkpoint。显然，WAL也需要Checkpoint。

（2）将spark.streaming.receiver.writeAheadLog.enable 设置为true。

Spark Streaming 里的 WAL 框架由一组抽象类和一组基于文件的具体实现组成。

这里涉及的抽象类有WriteAheadLog、WriteAheadLogRecordHandle。

WriteAheadLog是多条log的集合，每条具体的log的引用就是一个WriteAheadLog RecordHandle。

源码3-85　WriteAheadLog片段

```
public abstract class WriteAheadLog {

  /**
   * Write the record to the log and return a record handle, which contains all the information
   * necessary to read back the written record. The time is used to the index the record,
   * such that it can be cleaned later. Note that implementations of this abstract class must
   * ensure that the written data is durable and readable (using the record handle) by the
   * time this function returns.
   */
  // 写入一条log，返回一个指向这条log的句柄引用
  abstract public WriteAheadLogRecordHandle write(ByteBuffer record, long time);


  /**
   * Read a written record based on the given record handle.
```

```
    */
    // 通过log 的句柄引用，读出这条 log
    abstract public ByteBuffer read(WriteAheadLogRecordHandle handle);

    /**
     * Read and return an iterator of all the records that have been written but not yet cleaned up.
     */
    // 读取全部log
    abstract public Iterator<ByteBuffer> readAll();

    /**
     * Clean all the records that are older than the threshold time. It can wait for
     * the completion of the deletion.
     */
    // 清理过时的log条目
    abstract public void clean(long threshTime, boolean waitForCompletion);

    /**
     * Close this log and release any resources.
     */
    // 关闭
    abstract public void close();
}
```

做WAL冷备的地方，会调用WriteAheadLog子类的write方法，返回写入的log的

WriteAheadLogRecordHandle子类类型的句柄。

源码3-86　WriteAheadLogRecordHandle片段

```
public abstract class WriteAheadLogRecordHandle implements java.io.Serializable {
  // Handle 是一个空接口，需要具体的子类定义真正的内容
}
```

再看看两个抽象类的基于文件的具体实现。

WriteAheadLog基于文件的具体实现是FileBasedWriteAheadLog。

源码3-87　FileBasedWriteAheadLog片段

```
private[streaming] class FileBasedWriteAheadLog(
    conf: SparkConf,
    logDirectory: String,
    hadoopConf: Configuration,
    rollingIntervalSecs: Int,
    maxFailures: Int,
    closeFileAfterWrite: Boolean
  ) extends WriteAheadLog with Logging {
  ...
}
```

FileBasedWriteAheadLog主要是进行回滚文件（rolling file）的管理，里面有具体实现的write、read等方法。

WriteAheadLogRecordHandle 基于文件的case类是 FileBasedWriteAheadLogSegment。

源码3-88 FileBasedWriteAheadLogSegment片段

```
/** Class for representing a segment of data in a write ahead log file */
private[streaming] case class FileBasedWriteAheadLogSegment(path: String, offset: Long, length: Int)
  extends WriteAheadLogRecordHandle
```

2. Checkpoint

Spark Core对RDD已经有Checkpoint机制，但除此以外，Spark Streaming还要有自己的Checkpoint机制。

Spark Streaming应用程序如果不手动停止，则将一直运行下去，在实际中应用程序一般是24小时×7天不间断运行的，因此Streaming必须对诸如系统错误、JVM出错等与程序逻辑无关的错误（failures）具有很强的弹性，具备一定的非应用程序出错的容错性。Spark Streaming的Checkpoint机制便是为此设计的，它将足够多的信息通过设置检查点保存到某些具备容错性的存储系统（如HDFS）上，以便出错时能够迅速恢复。有两种数据可以设置检查点：

（1）Metadata Checkpointing。

将流式计算的信息保存到具备容错性的存储上，如HDFS，Metadata Checkpointing适用于当Spark Streaming应用程序Driver所在的节点出错时能够恢复，元数据包括：

- 配置信息。创建Spark Streaming应用程序的配置信息。
- DStream操作。在Spark Streaming应用程序中定义的DStreaming操作。
- 未完成的batches。在列队中没有处理完的作业。

（2）Data Checkpointing。

将生成的RDD保存到外部可靠的存储当中，对于一些数据跨度为多个batch的有状态tranformation操作来说，Checkpoint非常有必要，因为在这些转换操作生成的RDD对前一RDD有依赖，随着时间的增加，依赖链可能会非常长，Checkpoint机制能够切断依赖链，将

中间的RDD周期性地Checkpoint到可靠存储当中，从而在出错时可以直接从检查点恢复。

具体来说，Metadata Checkpointing主要还是从Driver失败中恢复，而Data Checkpointing用于对有状态的转换操作（比如updateStateByKey、reduceByKeyAndWindow等，可参考相关章节对状态操作的分析）进行checkpointing。

所以启用checkpoint，必须满足以下任一前提条件：

（1）Spark Streaming应用程序担心Driver挂掉后再次重启的进度丢失。

（2）存在状态操作。

下面剖析源码。先看看Checkpoint类。

源码3-89 Checkpoint类片段

```
private[streaming]
class Checkpoint(ssc: StreamingContext, val checkpointTime: Time)
 extends Logging with Serializable {
 // 部署模式
 val master = ssc.sc.master
 // 应用程序名
 val framework = ssc.sc.appName
 // 依赖的jar包
 val jars = ssc.sc.jars
 // DStreamGraph
 val graph = ssc.graph
 // Checkpoint目录
 val checkpointDir = ssc.checkpointDir
 // 进行Checkpoint的时间间隔设置
 val checkpointDuration = ssc.checkpointDuration
 // 批次对应时间的数组
```

```
  val pendingTimes = ssc.scheduler.getPendingTimes().toArray
  // 所有Spark配置项及其值
    val sparkConfPairs = ssc.conf.getAll
    ...
}
```

Checkpoint类包含一些配置信息。其中的graph即DStreamGraph对象，是设置检查点的重要信息来源。

Checkpoint冷备就是把Checkpoint对象序列化写入文件。所以Checkpoint是Serializable类。以上列出了成员变量，其中的checkpointDir是保存数据和元数据的文件夹路径。

必须注意的是，在Spark Streaming应用程序挂掉后，如果重新编译Spark Streaming应用程序再运行，是不能从前面的失败中恢复的，因为重新编译会导致不能反序列化Checkpoint。

一般在进行Checkpoint冷备时，首先需在程序中加一行代码，如下所示：

```
streamingContext.checkpoint(checkpointDirectory)
```

另外，对于想从Driver故障中重启并恢复的应用，则需要在程序中做到以下两点：

（1）如果应用程序首次启动，将创建一个新的 StreamContext 实例，设置一个支持容错的、可靠的文件系统（如HDFS、S3等）目录保存Checkpoint数据。

（2）如果从Driver失败中重启并恢复，则必须从Checkpoint目录导入Checkpoint数据来重新创建StreamingContext实例。

上面两点可以通过StreamingContext.getOrCreate做到，示例代码如下：

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // new context
```

```
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
    ssc
}

// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext _)

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...

// Start the context
context.start()
context.awaitTermination()
```

以Spark自带的例程RecoverableNetworkCount为例进行说明。

源码3-90　RecoverableNetworkCount

```
...
/**
 * Counts words in text encoded with UTF8 received from the network every second. This example also
 * shows how to use lazily instantiated singleton instances for Accumulator and Broadcast so that
 * they can be registered on driver failures.
 *
```

```
 *  Usage: RecoverableNetworkWordCount <hostname> <port> <checkpoint-directory> <output-file>
 * <hostname> and <port> describe the TCP server that Spark Streaming would connect to receive
 * data. <checkpoint-directory> directory to HDFS-compatible file system which checkpoint data
 * <output-file> file to which the word counts will be appended
 *
 * <checkpoint-directory> and <output-file> must be absolute paths
 *
 * To run this on your local machine, you need to first run a Netcat server
 *
 *      `$ nc -lk 9999`
 *
 * and run the example as
 *
 *      `$ ./bin/run-example org.apache.spark.examples.streaming.RecoverableNetworkWordCount \
 *              localhost 9999 ~/checkpoint/ ~/out`
 *
 * If the directory ~/checkpoint/ does not exist (e.g. running for the first time), it will create
 * a new StreamingContext (will print "Creating new context" to the console). Otherwise, if
 * checkpoint data exists in ~/checkpoint/, then it will create StreamingContext from
 * the checkpoint data.
 *
 * Refer to the online documentation for more details.
 */
object RecoverableNetworkWordCount {


   def createContext(ip: String, port: Int, outputPath: String, checkpointDirectory: String)
```

```
: StreamingContext = {

// 程序第一次运行时会创建该条语句，如果应用程序失败，则会从Checkpoint中恢复，该条语句不会执行
println("Creating new context")
val outputFile = new File(outputPath)
if (outputFile.exists()) outputFile.delete()
val sparkConf = new SparkConf().setAppName("RecoverableNetworkWordCount").setMaster("local[4]")
// Create the context with a 1 second batch size
val ssc = new StreamingContext(sparkConf, Seconds(1))
ssc.checkpoint(checkpointDirectory)

// 将socket作为数据源
val lines = ssc.socketTextStream(ip, port)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.foreachRDD((rdd: RDD[(String, Int)], time: Time) => {
  val counts = "Counts at time "+ time + ""+ rdd.collect().mkString("[", ", ", "]")
  println(counts)
  println("Appending to " + outputFile.getAbsolutePath)
  Files.append(counts + "\n", outputFile, Charset.defaultCharset())
})
ssc
}
// 将String转换成Int
private object IntParam {
def unapply(str: String): Option[Int] = {
```

```
    try {
      Some(str.toInt)
    } catch {
      case e: NumberFormatException => None
    }
  }
}
  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println("You arguments were "+ args.mkString("[", ", ", "]"))
      System.err.println(
        """
          |Usage: RecoverableNetworkWordCount <hostname> <port> <checkpoint-directory>
          |        <output-file>. <hostname> and <port> describe the TCP server that Spark
          |         Streaming would connect to receive data. <checkpoint-directory> directory to
          |         HDFS-compatible file system which checkpoint data <output-file> file to which the
          |         word counts will be appended
          |
          |In local mode, <master> should be 'local[n]' with n > 1
          |Both <checkpoint-directory> and <output-file> must be absolute paths
        """.stripMargin
      )
      System.exit(1)
    }
   val Array(ip, IntParam(port), checkpointDirectory, outputPath) = args
```

```
    // getOrCreate方法，从Checkpoint中重新创建StreamingContext对象或StreamingContext对象
    val ssc = StreamingContext.getOrCreate(checkpointDirectory,
      () => {
        createContext(ip, port, outputPath, checkpointDirectory)
      })
    ssc.start()
    ssc.awaitTermination()
  }
}
```

注意前面注释部分的命令行例子及其解释。

利用getOrCreate，创建了一个可恢复的StreamingContext。

读者可以先运行该程序，接着手动将程序停止，然后重新运行该程序，会注意到“Creating new context”信息不会再次在控制台显示。程序从Checkpoint目录中读取元数据信息，进行StreamingContext的恢复。

先简单介绍一下Checkpoint恢复流程的主要过程：

（1）调用StreamingContext.getOrCreate，使用CheckpointReader.read从文件中反序列化出Checkpoint对象，并使用Checkpoint对象去初始化StreamingContext对象。

（2）在StreamingContext中调用cp_.graph.restoreCheckpointData来恢复每个DStream.generatedRDDs。

（3）在JobGenerator中调用restart，重新提交那些在检查点保存中未被提交的Job。

现在开始，仍以源码3-89的RecoverableNetworkCount为例，进行Driver失败并重启的Checkpoint恢复流程分析。

先看StreamingContext.getOrCreate。

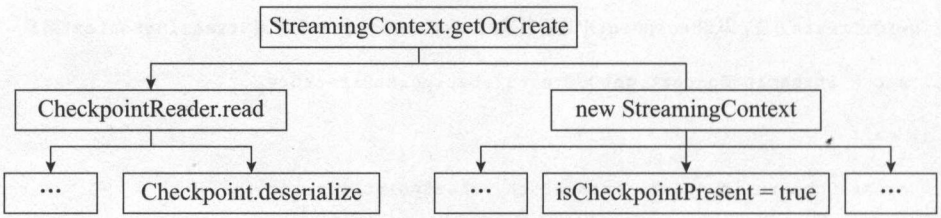


图3-16　Checkpoint的恢复流程

源码3-91　StreamingContext.getOrCreate

```
def getOrCreate(
    checkpointPath: String,
    creatingFunc: () => StreamingContext,
    hadoopConf: Configuration = SparkHadoopUtil.get.conf,
    createOnError: Boolean = false
  ): StreamingContext = {
  val checkpointOption = CheckpointReader.read(
    checkpointPath, new SparkConf(), hadoopConf, createOnError)
  checkpointOption.map(new StreamingContext(null, _, null)).getOrElse(creatingFunc())
}
```

源码3-92　CheckpointReader.read

```
  def read(
      checkpointDir: String,
      conf: SparkConf,
      hadoopConf: Configuration,
      ignoreReadError: Boolean = false): Option[Checkpoint] = {
```

```
val checkpointPath = new Path(checkpointDir)

// TODO(rxin): Why is this a def?!
def fs: FileSystem = checkpointPath.getFileSystem(hadoopConf)

// Try to find the checkpoint files
val checkpointFiles = Checkpoint.getCheckpointFiles(checkpointDir, Some(fs)).reverse
if (checkpointFiles.isEmpty) {
  return None
}

// Try to read the checkpoint files in the order
logInfo("Checkpoint files found: "+ checkpointFiles.mkString(","))
var readError: Exception = null
checkpointFiles.foreach(file => {
  logInfo("Attempting to load checkpoint from file " + file)
  try {
    val fis = fs.open(file)
    val cp = Checkpoint.deserialize(fis, conf)
    logInfo("Checkpoint successfully loaded from file " + file)
    logInfo("Checkpoint was generated at time " + cp.checkpointTime)
    return Some(cp)
  } catch {
    case e: Exception =>
      readError = e
      logWarning("Error reading checkpoint from file " + file, e)
```

```
      }
    })

    // If none of checkpoint files could be read, then throw exception
    if (!ignoreReadError) {
      throw new SparkException(
        s"Failed to read checkpoint from directory $checkpointPath", readError)
    }
    None
  }
}
```

因为是重启后运行，文件夹checkpointPath已经存在，故会调用Checkpoint.deserialize来做反序列化。

源码3-93 Checkpoint.deserialize

```
/** Deserialize a checkpoint from the input stream, or throw any exception that occurs */
def deserialize(inputStream: InputStream, conf: SparkConf): Checkpoint = {
  val compressionCodec = CompressionCodec.createCodec(conf)
  var ois: ObjectInputStreamWithLoader = null
  Utils.tryWithSafeFinally {

    // ObjectInputStream uses the last defined user-defined class loader in the stack
    // to find classes, which maybe the wrong class loader. Hence, a inherited version
```

```
    // of ObjectInputStream is used to explicitly use the current thread's default class
    // loader to find and load classes. This is a well know Java issue and has popped up
    // in other places (e.g., http://jira.codehaus.org/browse/GROOVY-1627)
    val zis = compressionCodec.compressedInputStream(inputStream)
    ois = new ObjectInputStreamWithLoader(zis,
      Thread.currentThread().getContextClassLoader)
    val cp = ois.readObject.asInstanceOf[Checkpoint]
    cp.validate()
    cp
  } {
    if (ois != null) {
      ois.close()
    }
  }
}
```

deserialize从输入流中反序列化一个Checkpoint对象并返回。

回到StreamingContext.getOrCreate（源码3-91）。调用checkpointOption.map(new StreamingContext(null, _, null))，重新生成了StreamingContext实例，其第二个参数是反序列化生成的Checkpoint对象。

源码3-94　StreamingContext片段

```
private[streaming] val isCheckpointPresent = (cp_ != null)
```

显然，由于Checkpoint变量cp_不为null，故isCheckpointPresent为true。isCheckpoint Present决定了StreamingContext中的一系列Checkpoint恢复。

源码3-95 StreamingContext.sc

```
private[streaming] val sc: SparkContext = {
  if (sc_ != null) {
     sc_
  } else if (isCheckpointPresent) {
     SparkContext.getOrCreate(cp_.createSparkConf())
  } else {
     throw new SparkException("Cannot create StreamingContext without a SparkContext")
  }
}
```

如果isCheckpointPresent为true，就会从Checkpoint中恢复SparkContext。

源码3-96 StreamingContext.graph

```
private[streaming] val graph: DStreamGraph = {
  if (isCheckpointPresent) {
     cp_.graph.setContext(this)
     cp_.graph.restoreCheckpointData()
     cp_.graph
  } else {
     require(batchDur_ != null, "Batch duration for StreamingContext cannot be null")
     val newGraph = new DStreamGraph()
```

```
    newGraph.setBatchDuration(batchDur_)
    newGraph
  }
}
```

以上代码恢复了DStreamGraph，看DStreamGraph.restoreCheckpointData。

源码3-97　DStreamGraph.restoreCheckpointData

```
def restoreCheckpointData() {
  logInfo("Restoring checkpoint data")
  this.synchronized {
    outputStreams.foreach(_.restoreCheckpointData())
  }
  logInfo("Restored checkpoint data")
}
```

DStreamGraph.restoreCheckpointData调用了DStream.restoreCheckpointData。

源码3-98　DStream.restoreCheckpointData

```
private[streaming] def restoreCheckpointData() {
  if (!restoredFromCheckpointData) {
    // Create RDDs from the checkpoint data
    logInfo("Restoring checkpoint data")
    checkpointData.restore()
    dependencies.foreach(_.restoreCheckpointData())
```

```
    restoredFromCheckpointData = true
    logInfo("Restored checkpoint data")
  }
}
```

DStream.restoreCheckpointData调用checkpointData.restore来恢复各DStream的generatedRDDs。针对依赖的DStream会递归调用restoreCheckpointData。下面看看DStream的generatedRDDs的来源。

源码3-99 DStreamCheckpointData.restore

```
/**
 * Restore the checkpoint data. This gets called once when the DStream graph
 * (along with its DStreams) are being restored from a graph checkpoint file.
 * Default implementation restores the RDDs from their checkpoint files.
 */
def restore() {
  // Create RDDs from the checkpoint data
  currentCheckpointFiles.foreach {
    case(time, file) => {
      logInfo("Restoring checkpointed RDD for time "+ time + "from file '" + file + "'")
      dstream.generatedRDDs += ((time, dstream.context.sparkContext.checkpointFile[T](file)))
    }
  }
}
```

DStream的generatedRDDs来源于checkpointFile。来看看这个checkpointFile的来源。

源码3-100　SparkContext.checkpointFile

```
protected[spark] def checkpointFile[T: ClassTag](path: String): RDD[T] = withScope {
  new ReliableCheckpointRDD[T](this, path)
}
```

生成了ReliableCheckpointRDD对象。

源码3-101　ReliableCheckpointRDD.compute

```
  /**
   * Read the content of the checkpoint file associated with the given partition.
   */
  override def compute(split: Partition, context: TaskContext): Iterator[T] = {
    val file = new Path(checkpointPath, ReliableCheckpointRDD.checkpointFileName(split.index))
    ReliableCheckpointRDD.readCheckpointFile(file, broadcastedConf, context)
  }
```

DStream的generatedRDDs确实来源于checkpointPath 指定的Checkpoint文件。

isCheckpointPresent为true，也使JobGenerator对象在启动时调用了restart方法。

源码3-102　JobGenerator.restart

```
/** Restarts the generator based on the information in checkpoint */
private def restart() {
  // If manual clock is being used for testing, then
```

```
// either set the manual clock to the last checkpointed time,
// or if the property is defined set it to that time
if (clock.isInstanceOf[ManualClock]) {
  val lastTime = ssc.initialCheckpoint.checkpointTime.milliseconds
  val jumpTime = ssc.sc.conf.getLong("spark.streaming.manualClock.jump", 0)
  clock.asInstanceOf[ManualClock].setTime(lastTime + jumpTime)
}

val batchDuration = ssc.graph.batchDuration

// Batches when the master was down, that is,
// between the checkpoint and current restart time
val checkpointTime = ssc.initialCheckpoint.checkpointTime
val restartTime = new Time(timer.getRestartTime(graph.zeroTime.milliseconds))
val downTimes = checkpointTime.until(restartTime, batchDuration)
logInfo("Batches during down time (" + downTimes.size + " batches): "
  + downTimes.mkString(", "))

// Batches that were unprocessed before failure
// 先恢复上一次Application执行时已经产生但是还没有成功执行完成的Streaming Job
val pendingTimes = ssc.initialCheckpoint.pendingTimes.sorted(Time.ordering)
logInfo("Batches pending processing ("+ pendingTimes.size + "batches): "+
  pendingTimes.mkString(", "))
// Reschedule jobs for these times
// 补上从崩溃到重新执行的时间之间没有产生的Job
```

```
  val timesToReschedule = (pendingTimes ++ downTimes).filter { _ < restartTime }
    .distinct.sorted(Time.ordering)
  logInfo("Batches to reschedule ("+ timesToReschedule.size + "batches): "+
    timesToReschedule.mkString(", "))
  timesToReschedule.foreach { time =>
    // Allocate the related blocks when recovering from failure, because some blocks that were
    // added but not allocated, are dangling in the queue after recovering, we have to allocate
    // those blocks to the next batch, which is the batch they were supposed to go.
    jobScheduler.receiverTracker.allocateBlocksToBatch(time)
    // allocate received blocks to batch
    // 先执行丢失的Job
  jobScheduler.submitJobSet(JobSet(time, graph.generateJobs(time)))
  }

  // Restart the timer
  // 产生Application重启时刻开始的新Job
  timer.start(restartTime.milliseconds)
  logInfo("Restarted JobGenerator at " + restartTime)
}
```

JobGenerator的restart方法会重新提交那些在cp_中未被提交的jobs，然后把从崩溃到重新执行的时间之间没有产生的Job也补上，并让Spark先执行这些丢失的Job。最后执行重启时开始的新Job。

这些Checkpoint恢复保证了Spark Streaming恢复到出异常前的状态，使后续没有被处理的Job不被漏掉。

需要注意的是，异常后，如果做了重新编译再重启，则Checkpoint中的DStreamGraph会过时，反序列化会失败。

上面对备份的原理、冷备的恢复流程进行了分析。下面对各具体的备份流程进行分析。

3.5.2 Driver容错机制

Driver负责Spark Streaming的调度管理。一旦Driver失效，Spark Streaming的调度管理便失效，Spark Streaming就不能保证正常运行。所以只能是做好日常冷备，在有故障时，重启Spark Streaming应用程序来恢复运行。

本节介绍以下用于备份的类：

- ReceivedBlockTracker，用于WAL冷备。
- DStream和JobGenerator，用于Checkpoint冷备。

1. ReceivedBlockTracker中的WAL冷备

Block数据的元信息（metadata）是上报到ReceiverTracker，然后交给ReceivedBlockTracker 做具体管理的。ReceivedBlockTracker采用WAL冷备方式进行备份，在Driver失效后，新的ReceivedBlockTracker会读取WAL并恢复Block的元信息。

源码3-103 ReceivedBlockTracker.addBlock

```
/** Add received block. This event will get written to the write ahead log (if enabled). */
def addBlock(receivedBlockInfo: ReceivedBlockInfo): Boolean = {
  try {
    val writeResult = writeToLog(BlockAdditionEvent(receivedBlockInfo))
    ...
    writeResult
  } catch {
```

```
    ...
  }
}
```

ReceivedBlockTracker.addBlock是在数据接收过程中更新当前接收且未被分配给Job的Block时被调用，可参考源码3-48及其上下文。

源码3-104　ReceivedBlockTracker.allocateBlocksToBatch

```
def allocateBlocksToBatch(batchTime: Time): Unit = synchronized {
  if (lastAllocatedBatchTime == null || batchTime > lastAllocatedBatchTime) {
    ...
    if (writeToLog(BatchAllocationEvent(batchTime, allocatedBlocks))) {
      ...
    } else {
      ...
    }
  } else {
    ...
  }
}
```

ReceivedBlockTracker.allocateBlocksToBatch是在数据处理过程中在JobGenerator生成Jobs之前被调用的，因为Job的生成中会有相应的Block分配给Job，可参考源码3-55及其上下文。

源码3-105 ReceivedBlockTracker.cleanupOldBatches

```
def cleanupOldBatches(cleanupThreshTime: Time, waitForCompletion: Boolean): Unit = synchronized {
  require(cleanupThreshTime.milliseconds < clock.getTimeMillis())
  val timesToCleanup = timeToAllocatedBlocks.keys.filter { _ < cleanupThreshTime }.toSeq
  logInfo("Deleting batches " + timesToCleanup)
  if (writeToLog(BatchCleanupEvent(timesToCleanup))) {
    timeToAllocatedBlocks --= timesToCleanup
    writeAheadLogOption.foreach(_.clean(cleanupThreshTime.milliseconds, waitForCompletion))
  } else {
    logWarning("Failed to acknowledge batch clean up in the Write Ahead Log.")
  }
}
```

上面的ReceivedBlockTracker的3段代码都调用了writeToLog，分别把BlockAdditionEvent、BatchAllocationEvent、BatchCleanupEvent这3种消息保存到了WAL里。

源码3-106 ReceivedBlockTracker.writeToLog

```
/** Write an update to the tracker to the write ahead log */
private def writeToLog(record: ReceivedBlockTrackerLogEvent): Boolean = {
  if (isWriteAheadLogEnabled) {
    logTrace(s"Writing record: $record")
    try {
      writeAheadLogOption.get.write(ByteBuffer.wrap(Utils.serialize(record)),
        clock.getTimeMillis())
      true
```

```
    } catch {
      case NonFatal(e) =>
        logWarning(s"Exception thrown while writing record: $record to the WriteAheadLog.", e)
        false
    }
  } else {
    true
  }
}
```

如果系统出现异常，需要从WAL恢复，就能够获得这3种日志信息，并从头开始重新构建出ReceivedBlockTracker对象及其状态成员。

2. DStream和JobGenerator的Checkpoint冷备

Driver端用Checkpoint冷备来做容错，需要对数据、元数据进行备份。数据主要指从出错时开始恢复处理所涉及的数据。元数据包括Spark Steaming的程序配置、数据处理的逻辑等。

Checkpoint通常落地到可靠存储（如HDFS）中。Checkpoint发起的默认时间间隔和BatchDuration一致，即每次Batch发起、提交了需要运行的Job后就做 Checkpoint，另外在Job完成了更新任务状态的时候再次做Checkpoint。

Driver端的Checkpoint冷备发生在两处：

（1）JobGenerator.generateJobs（源码3-55）中的提交时间间隔内的所有Job后。

（2）JobGenerator.clearMetadata（源码3-78）中满足shouldCheckpoint为true的条件时。

这两处都发送了DoCheckpoint消息。

下面做具体分析。

先给出DoCheckpoint消息的处理流程图。

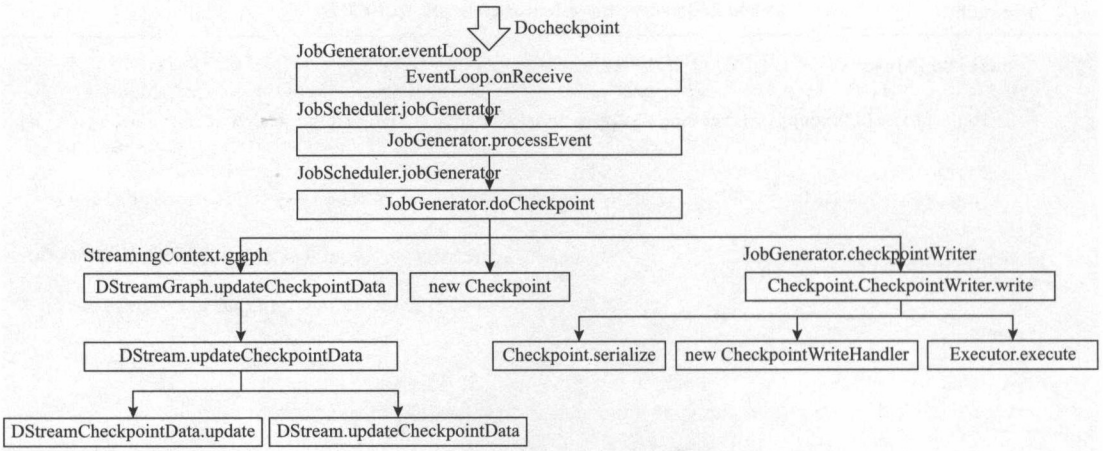


图3-17　处理DoCheckpoint消息的流程

以上的流程图主要体现了两件事情：

（1）在设置检查点前，先把当前批次的元数据放入DStreamGraph对象中。

（2）设置检查点。生成Checkpoint对象，记入数据和元数据，并把Checkpoint对象序列化到Checkpoint文件中。

JobGenerator的eventLoop会使用EventLoop.onReceive来处理消息。在onReceive中指定了用JobGenerator.processEvent。

源码3-107　JobGenerator.processEvent

```
/** Processes all events */
private def processEvent(event: JobGeneratorEvent) {
  logDebug("Got event " + event)
  event match {
    case GenerateJobs(time) => generateJobs(time)
    case ClearMetadata(time) => clearMetadata(time)
    case DoCheckpoint(time, clearCheckpointDataLater) =>
      doCheckpoint(time, clearCheckpointDataLater)
```

```
    case ClearCheckpointData(time) => clearCheckpointData(time)
  }
}
```

在模式匹配消息DoCheckpoint后，调用JobGenerator.doCheckpoint，做相应Checkpoint冷备。

源码3-108　JobGenerator.doCheckpoint

```
/** Perform checkpoint for the give `time`. */
private def doCheckpoint(time: Time, clearCheckpointDataLater: Boolean) {
  if (shouldCheckpoint && (time - graph.zeroTime).isMultipleOf(ssc.checkpointDuration)) {
    logInfo("Checkpointing graph for time " + time)
    // 将输出流中的每个DStream信息转化成相应的Checkpoint信息
    ssc.graph.updateCheckpointData(time)
    checkpointWriter.write(new Checkpoint(ssc, time), clearCheckpointDataLater)
  }
}
```

先分析ssc.graph.updateCheckpointData。

源码3-109　DStreamGraph.updateCheckpointData

```
def updateCheckpointData(time: Time) {
  logInfo("Updating checkpoint data for time " + time)
  this.synchronized {
    outputStreams.foreach(_.updateCheckpointData(time))
```

```
    }
    logInfo("Updated checkpoint data for time " + time)
  }
```

在outputStreams中的每个DStream都调用了DStream.updateCheckpointData。

源码3-110 DStream.updateCheckpointData

```
/**
 * Refresh the list of checkpointed RDDs that will be saved along with checkpoint of
 * this stream. This is an internal method that should not be called directly. This is
 * a default implementation that saves only the file names of the checkpointed RDDs to
 * checkpointData. Subclasses of DStream (especially those of InputDStream) may override
 * this method to save custom checkpoint data.
 */
private[streaming] def updateCheckpointData(currentTime: Time) {
  logDebug("Updating checkpoint data for time " + currentTime)
  checkpointData.update(currentTime)
  dependencies.foreach(_.updateCheckpointData(currentTime))
  logDebug("Updated checkpoint data for time "+ currentTime + ": "+ checkpointData)
}
```

DStreamCheckpointData类型的成员checkpointData会被更新。DStreamCheckpointData用来记录各批次所对应的RDD的Checkpoint文件。读者可能注意到，以下有成员包含关系的类都是Serializable子类：Checkpoint－DStreamGraph－DStream－DStreamCheckpointData。

这些都是为最终进行Checkpoint冷备时能做序列化而设计的。

继续看DStreamCheckpointData的update。

源码3-111　DStreamCheckpointData.update

```
/**
 * Updates the checkpoint data of the DStream. This gets called every time
 * the graph checkpoint is initiated. Default implementation records the
 * checkpoint files to which the generate RDDs of the DStream has been saved.
 */
def update(time: Time) {

  // Get the checkpointed RDDs from the generated RDDs
  val checkpointFiles = dstream.generatedRDDs.filter(_._2.getCheckpointFile.isDefined)
                                             .map(x => (x._1, x._2.getCheckpointFile.get))
  logDebug("Current checkpoint files:\n" + checkpointFiles.toSeq.mkString("\n"))

  // Add the checkpoint files to the data to be serialized
  if (!checkpointFiles.isEmpty) {
    currentCheckpointFiles.clear()
    currentCheckpointFiles ++= checkpointFiles
    // Add the current checkpoint files to the map of all checkpoint files
    // This will be used to delete old checkpoint files
    timeToCheckpointFile ++= currentCheckpointFiles
    // Remember the time of the oldest checkpoint RDD in current state
    timeToOldestCheckpointFileTime(time) = currentCheckpointFiles.keys.min(Time.ordering)
  }
}
```

记录了当前批次生成的RDD的Checkpoint文件路径。

源码3-110（DStream.updateCheckpointData）中，当前DStream的每个被依赖的DStream又会调用DStream.updateCheckpointData，所以形成了递归调用，一直回溯到InputDStream。

回到源码3-108的JobGenerator.doCheckpoint，它会调用CheckpointWriter.write，其参数有新生成的Checkpoint对象。

源码3-112 Checkpoint.CheckpointWriter.write

```
def write(checkpoint: Checkpoint, clearCheckpointDataLater: Boolean) {
  try {
    val bytes = Checkpoint.serialize(checkpoint, conf)
    executor.execute(new CheckpointWriteHandler(
      checkpoint.checkpointTime, bytes, clearCheckpointDataLater))
    logInfo("Submitted checkpoint of time "+ checkpoint.checkpointTime + "writer queue")
  } catch {
    case rej: RejectedExecutionException =>
      logError("Could not submit checkpoint task to the thread pool executor", rej)
  }
}
```

调用Checkpoint.serialize把Checkpoint对象序列化，并在Executor线程池中用线程来新建CheckpointWriteHandler对象。新建CheckpointWriteHandler对象时，使用了Checkpoint对象序列化生成的字节数组作为参数。Checkpoint的数据是通过CheckpointWriter真正地写入到外部存储系统中。

3.5.3 Executor容错机制

有的Executor接收数据，有的Executor处理数据。Spark集群有多个Executor，Executor失效时，是可以被替换的。所以除了冷备，也有热备，通过Executor替换就能保证Spark Core的Job继续不间断进行下去。

本节介绍以下用于备份的类：

- Receiver子类、BlockManager，用于热备。
- WriteAheadLogBasedBlockHandler，用于WAL冷备。

1. Reciever子类和BlockManager中的热备

热备是指在存储块数据时，将其存储到本Executor，同时复制到另外一个Executor上去。这样在一个replica失效后，可以立刻无感知切换到另一份replica进行计算。热备基本是使用Spark Core的容错机制。

Receiver类本身就有指明存储级别的参数StorageLevel。Receiver子类也会有。在实现自己的Receiver时，指定一下StorageLevel为MEMORY_ONLY_2 或MEMORY_AND_DISK_2就可以了。其中的2表示数据备份到集群中两个不同的节点。

比如定制自己的Receiver子类MyReciever。

源码3-113 MyReciever

```
class MyReceiver extends Receiver(StorageLevel.MEMORY_ONLY_2) {

  override def onStart(): Unit = {}

  override def onStop(): Unit = {}

}
```

这样，Receiver在将数据store给ReceiverSupervisorImpl的时候，将同时指明此storageLevel。ReceiverSupervisorImpl也将根据此storageLevel将块数据具体地存储给BlockManager。

然后就是依靠BlockManager进行热备。下面以ReceiverSupervisorImpl向Spark Core中的BlockManager存储一个byteBuffer为例对此加以说明。

源码3-114 BlockManager.putBytes

```
/**
 * Put a new block of serialized bytes to the block manager.
 * Return a list of blocks updated as a result of this put.
 */
def putBytes(
    blockId: BlockId,
    bytes: ByteBuffer,
    level: StorageLevel,
    tellMaster: Boolean = true,
    effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId, BlockStatus)] = {
  require(bytes != null, "Bytes is null")
  doPut(blockId, ByteBufferValues(bytes), level, tellMaster, effectiveStorageLevel)
}
```

BlockManager.putBytes时，实际是直接调用doPut。

源码3-115 BlockManager.doPut

```
private def doPut(blockId: BlockId, data: BlockValues, level: StorageLevel, ...)
  : Seq[(BlockId, BlockStatus)] = {
  ...
  // 如果putLevel.replication > 1，则定义这个 future，复制数据到另外的Executor上
```

```
val replicationFuture = data match {
  case b: ByteBufferValues if putLevel.replication > 1 =>
    val bufferView = b.buffer.duplicate()
    Future {
      // future启动时调用replicate，复制数据到另外的Executor上
      replicate(blockId, bufferView, putLevel)
    }(futureExecutionContext)
  case _ => null
}

putBlockInfo.synchronized {
  ...
  // 存储到本机BlockManager的blockStore里
  val result = data match {
    case IteratorValues(iterator) =>
      blockStore.putIterator(blockId, iterator, putLevel, returnValues)
    case ArrayValues(array) =>
      blockStore.putArray(blockId, array, putLevel, returnValues)
    case ByteBufferValues(bytes) =>
      bytes.rewind()
      blockStore.putBytes(blockId, bytes, putLevel)
  }
}

// 再次判断是否putLevel.replication > 1
if (putLevel.replication > 1) {
```

```
    data match {
      case ByteBufferValues(bytes) =>
        // 如果之前已启动了replicate的future，那么这里就同步地等这个future结束
        if (replicationFuture != null) {
          Await.ready(replicationFuture, Duration.Inf)
        }
      case _ =>
        val remoteStartTime = System.currentTimeMillis
        if (bytesAfterPut == null) {
          if (valuesAfterPut == null) {
            throw new SparkException(
              "Underlying put returned neither an Iterator nor bytes! This shouldn't happen.")
          }
          bytesAfterPut = dataSerialize(blockId, valuesAfterPut)
        }
        // 否则之前没有启动replicate的future，那么这里就同步地调用replicate方法
        // 复制数据到另外的Executor上
        replicate(blockId, bytesAfterPut, putLevel)
        logDebug("Put block %s remotely took %s"
          .format(blockId, Utils.getUsedTimeMs(remoteStartTime)))
    }
  }

  ...
}
```

可以看到，如果指定需要replicate，那么当putBytes方法返回时，除了存储到本机，还会复制到另外的Executor上。对于BlockManager的putIterator也是同样的语义，因为BlockManager的putIterator和BlockManager的putBytes一样，都是基于BlockManager 的doPut来实现的。

总结一下，Receiver收到的数据，通过ReceiverSupervisorImpl将数据交给BlockManager存储；BlockManager本身支持用replicate将数据复制到另外的Executor上，这样就完成了Receiver源头数据的热备过程。

在计算时，如果一个Executor失效导致一份数据丢失，那么计算任务将转而从另一个Executor上的同一份数据获取数据。由于另一份块数据是现成的，不需要像冷备那样重新读取，所以这里不会有恢复时间。

2. WriteAheadLogBasedBlockHandler中的WAL冷备

在3.2节中已经分析过，Receiver 收到的数据以 Block（数据块）的形式存储。看源码3-43可知道，其中receivedBlockHandler既可能是WriteAheadLogBasedBlockHandler对象，也可能是BlockManagerBasedBlockHandler对象。这会由应用程序运行时的配置来决定。这个配置项是spark.streaming.receiver.writeAheadLog.enable。如果设置为true，就使用了WriteAheadLogBasedBlockHandler对象。WriteAheadLogBasedBlockHandler.storeBlock在存储Block时被调用。

源码3-116 WriteAheadLogBasedBlockHandler.storeBlock

```
/**
 * This implementation stores the block into the block manager as well as a write ahead log.
 * It does this in parallel, using Scala Futures, and returns only after the block has
 * been stored in both places.
 */
def storeBlock(blockId: StreamBlockId, block: ReceivedBlock): ReceivedBlockStoreResult = {
```

```
var numRecords = None: Option[Long]
// Serialize the block so that it can be inserted into both
val serializedBlock = block match {
  case ArrayBufferBlockarrayBuffer() =>
    numRecords = Some(arrayBuffer.size.toLong)
    blockManager.dataSerialize(blockId, arrayBuffer.iterator)
  case IteratorBlock(iterator) =>
    val countIterator = new CountingIterator(iterator)
    val serializedBlock = blockManager.dataSerialize(blockId, countIterator)
    numRecords = countIterator.count
    serializedBlock
  case ByteBufferBlock(byteBuffer) =>
    byteBuffer
  case _ =>
    throw new Exception(s"Could not push $blockId to block manager, unexpected block type")
}

// Store the block in block manager
val storeInBlockManagerFuture = Future {
  val putResult =
    blockManager.putBytes(blockId, serializedBlock, effectiveStorageLevel, tellMaster = true)
  if (!putResult.map { _._1 }.contains(blockId)) {
    throw new SparkException(
      s"Could not store $blockId to block manager with storage level $storageLevel")
  }
}
```

```
    // Store the block in write ahead log
    val storeInWriteAheadLogFuture = Future {
      writeAheadLog.write(serializedBlock, clock.getTimeMillis())
    }

    // Combine the futures, wait for both to complete, and return the write ahead log record handle
    val combinedFuture = storeInBlockManagerFuture.zip(storeInWriteAheadLogFuture).map(_._2)
    val walRecordHandle = Await.result(combinedFuture, blockStoreTimeout)
    WriteAheadLogBasedStoreResult(blockId, numRecords, walRecordHandle)
  }
```

WriteAheadLogBasedBlockHandler.storeBlock保存Block后，通过代码writeAheadLog.write进行WAL冷备。

3.6 No Receiver方式

目前所讲述的数据接收涉及的InputDStream主要是使用ReceiverInputDStream，是针对Receiver方式开展的剖析。

在企业级Spark Streaming应用程序开发中越来越多地采用No Receiver的方式。No Receiver方式有自己的优势，比如更大的控制的自由度、语义一致性等。所以对No Receiver方式和Receiver方式都需要进一步研究、思考。

其实No Receiver方式更符合操作、处理数据的思路。作为计算框架的Spark，底层会

有数据来源不使用Receiver，直接操作数据源，是更自然的方式。操作数据来源的封装器一定是RDD类型的。

Apache Kafka提供了基于Receiver和No Receiver的Direct方式，下面详细说明。

（1）基于Receiver。

这种方式是将Kafka Consumer的偏移管理交给Kafka，偏移量会保存在ZooKeeper里，失效后由Kafka基于offset进行重放。ZooKeeper是一个分布式的、开放源码的应用程序协调服务框架，此处用于保存Kafka集群的元数据。

这样可能出现的问题是，Kafka 将同一个offset的数据重放给两个batch实例。从而只能保证at-least once（至少一次）的语义。

（2）Direct方式，不基于Receiver。

Spark Streaming中为了封装推出了KafkaRDD，只不过针对不同来源的数据定制了相应的RDD。

下面分析Kafka的Direct方式。先从应用程序代码开始。

源码3-117　使用Direct方式的应用程序代码片段

```
object DStreamExample {
  def main(args: Array[String]): Unit = {
    val sparkConf = new SparkConf().setAppName("example")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    val kafkaParams = Map[String, String](
      "metadata.broker.list" -> "1.1.1.1",
      "group.id" -> "123",
      "auto.offset.reset" -> "smallest"
    )
   val topicsSet = HashSet("test")
```

```
    val messages = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder]
(ssc, kafkaParams, topicsSet)
    val message2 = messages.map(line => line + "111")
      .filter(line => line.length > 10)
    message2.foreachRDD(rdd => rdd.collect)
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Direct方式是利用KafkaUtils.createDirectStream来生成InputDStream。

源码3-118　KafkaUtils.createDirectStream

```
def createDirectStream[
  K: ClassTag,
  V: ClassTag,
  KD <: Decoder[K]: ClassTag,
  VD <: Decoder[V]: ClassTag] (
    ssc: StreamingContext,
    kafkaParams: Map[String, String],
    topics: Set[String]
): InputDStream[(K, V)] = {
  val messageHandler = (mmd: MessageAndMetadata[K, V]) => (mmd.key, mmd.message)
  val kc = new KafkaCluster(kafkaParams)
  val fromOffsets = getFromOffsets(kc, kafkaParams, topics)
```

```
  new DirectKafkaInputDStream[K, V, KD, VD, (K, V)](
    ssc, kafkaParams, fromOffsets, messageHandler)
}
```

KafkaUtils.createDirectStream会生成一个DirectKafkaInputDStream对象。

每个DirectKafkaInputDStream 会持有一个KafkaCluster实例。KafkaCluster 这个类是实际负责和Kafka 交互的类，该类会获取Kafka的Partition信息。

每个Kafka的Topic/Partition对应一个RDD Partition。

源码3-119 DirectKafkaInputDStream

```
class DirectKafkaInputDStream[
  K: ClassTag,
  V: ClassTag,
  U <: Decoder[K]: ClassTag,
  T <: Decoder[V]: ClassTag,
  R: ClassTag](
    ssc_ : StreamingContext,
    val kafkaParams: Map[String, String],
    val fromOffsets: Map[TopicAndPartition, Long],
    messageHandler: MessageAndMetadata[K, V] => R
  ) extends InputDStream[R](ssc_) with Logging {
  // kafka默认的最大重试次数为一次，以确保语义一致性
  val maxRetries = context.sparkContext.getConf.getInt(
    "spark.streaming.kafka.maxRetries", 1)
...
```

Direct方式的Spark Streaming在生成Job时一定会用上DirectKafkaInputDStream.compute。

源码3-120　DirectKafkaInputDStream.compute

```
override def compute(validTime: Time): Option[KafkaRDD[K, V, U, T, R]] = {
    val untilOffsets = clamp(latestLeaderOffsets(maxRetries))
    val rdd = KafkaRDD[K, V, U, T, R](
      context.sparkContext, kafkaParams, currentOffsets, untilOffsets, messageHandler)

    // Report the record number and metadata of this batch interval to
        InputInfoTracker.
    val offsetRanges = currentOffsets.map { case (tp, fo) =>
      val uo = untilOffsets(tp)
      OffsetRange(tp.topic, tp.partition, fo, uo.offset)
    }
    val description = offsetRanges.filter { offsetRange =>
      // Don't display empty ranges.
      offsetRange.fromOffset != offsetRange.untilOffset
    }.map { offsetRange =>
      s"topic: ${offsetRange.topic}\tpartition: ${offsetRange.partition}\t" +
        s"offsets: ${offsetRange.fromOffset} to ${offsetRange.untilOffset}"
    }.mkString("\n")
    // Copy offsetRanges to immutable.List to prevent from being modified by the user
    val metadata = Map(
      "offsets" -> offsetRanges.toList,
      StreamInputInfo.METADATA_KEY_DESCRIPTION -> description)
```

```
    val inputInfo = StreamInputInfo(id, rdd.count, metadata)
    ssc.scheduler.inputInfoTracker.reportInfo(validTime, inputInfo)

    currentOffsets = untilOffsets.map(kv => kv._1 -> kv._2.offset)
    Some(rdd)
  }
```

DirectKafkaInputDStream.compute主要做了以下操作：

（1）获取对应Kafka Partition的untilOffset。这样就确定了需要获取数据的区间，同时也就知道了需要计算多少数据了。

（2）构建一个KafkaRDD实例。这里可以看到，每个计算周期里，KafkaRDD的实例和DirectKafkaInputDStreamr的关系是一一对应的。每次compute就是产生一个KafkaRDD。

（3）将相关的offset信息报给InputInfoTracker。

（4）返回该RDD。

最后再看看KafkaRDD。

源码3-121 KafkaRDD片段

```
class KafkaRDD[
  K: ClassTag,
  V: ClassTag,
  U <: Decoder[_]: ClassTag,
  T <: Decoder[_]: ClassTag,
  R: ClassTag] private[spark] (
    sc: SparkContext,
    kafkaParams: Map[String, String],
```

```
    val offsetRanges: Array[OffsetRange],
    leaders: Map[TopicAndPartition, (String, Int)],
    messageHandler: MessageAndMetadata[K, V] => R
  ) extends RDD[R](sc, Nil) with Logging with HasOffsetRanges {
  override def getPartitions: Array[Partition] = {
    offsetRanges.zipWithIndex.map { case (o, i) =>
        val (host, port) = leaders(TopicAndPartition(o.topic, o.partition))
        new KafkaRDDPartition(i, o.topic, o.partition, o.fromOffset, o.untilOffset, host, port)
    }.toArray
  }
...
```

KafkaRDD本身包含多个KafkaRDDPartition，其实就是对应了多个Kafka的Partition。一个Partition只能属于一个Topic。KafkaRDD 包含的KafkaRDDPartition 个数就是Kafka的Partition个数，每个KafkaRDDPartition其实只是包含一些信息，如topic、offset等。

拉数据是通过KafkaRDDIterator来完成的，这里就不再剖析KafkaRDDIterator代码。一个KafkaRDDIterator对应一个 KafkaRDDPartition。整个过程都是延时过程，也就是数据其实都在Kafka中保存着，直到有实际的action被触发，才会又去Kafka主动拉数据。

通过分析代码可以知道，Spark Streaming通过Direct方式直接管理offset。可以给定offset范围，直接去Kafka的硬盘上读数据，使用Spark Streaming自身的均衡来代替Kafka做的均衡。这样可以保证，每个offset范围属于且只属于一个batch。

下面仍以Direct方式为例，详解Spark Streaming在源头数据失效后是如何从上游重放数据的。这里的实现分为两个层面：

（1）DirectKafkaInputDStream。负责侦测最新offset，并将offset分配至唯一个batch。会在每次batch生成时，依靠latestLeaderOffsets方法去侦测最新的offset，然后与上一个

batch侦测到的offset相减，就能得到一个offset的范围offsetRange，把这个offset范围内的数据唯一地分配到本 batch 来处理。

（2）KafkaRDD。负责去读指定offset范围内的数据，并基于此数据进行计算。会生成一个Kafka的SimpleConsumer。SimpleConsumer是Kafka最底层、直接对着 Kafka 硬盘上的文件读数据的类。如果Task失败，导致任务重新下发，那么offset范围仍然维持不变，将直接重新生成一个Kafka的SimpleConsumer去读数据。

所以Direct的方式，归根结底是由Spark Streaming框架来负责整个offset的侦测、batch分配和实际读取数据，并且这些分batch的信息都通过设置检查点存储起来（一般是HDFS）了。这里没有利用ZooKeeper来均衡consumer和记录offset的功能，而是把Kafka直接当成一个底层的文件系统来使用了。

下面分析一下exactly-once。

Receiver方式有这样一个问题：Receiver接收数据是积累到一定记录后才会写入WAL，如果Receiver线程失败，数据有可能会丢失。而Kafka的Direct方式，利用Kafka Direct API为Kafka数据源提供了exactly-once保证。但要实现端到端的exactly-once保证，还必须要求输出算子也能有exactly-once保证。对输出算子exactly-once保证，请看3.7节对输出不重复的分析。

Kafka的Direct方式作为重要的No Receiver方式，其好处有必要总结一下：

（1）没缓存，就没内存溢出。

（2）Receiver方式会和Worker的Executor绑定，不方便做分布式（当然已有技巧做到分布式了）。RDD的Direct方式可以容易地做到分布式。

（3）Receiver方式在流进来的数据不能得到及时处理、持续积累并延迟下去的话，Spark Streaming就有可能崩溃。Direct方式则不会出现这种情况，因为如果延迟了，就不会处理后面的数据流。

（4）完全的语义一致性，确保数据一定会消费，而且不会重复消费。

（5）Direct方式比Receiver方式性能高。根据自己的InputDStream进行配置，还可以设置很多DStream。

No Receiver方式要求上游数据源支持重放。当然，我们讲上游重放，并不只局限于Kafka，而是说凡是支持消息重放的上游都可以。比如，HDFS 也可以看作一个支持重放的可靠上游。FileInputDStream 就是利用重放的方式保证了Executor失效后的源头数据的可读性。

另外，数据接收的反压机制很重要。设置spark.streaming.backpressure.enabled为true，可以试探数据流进来的速度和当前的处理能力是否一致，如果不一致，可以动态调整资源。反压机制会在后面分析。

3.7 输出不重复

输出操作在默认情况下已保证了at-least once的语义，如果再做到输出不重复，那就能实现exactly-once语义。

为什么会有输出重复这个问题？因为Spark Streaming在计算的时候基于Spark Core，而Spark Core做Task重试、任务推测、Stage重复、Job重试等事情时，都会导致Spark Streaming的结果会部分地重复输出。

对应的系统配置上的解决方案有以下两个：

（1）Task重试、Stage重复、Job重试所导致的重复输出。这些就是Job 失败，设置spark.task.maxFailures次数为1。如果使用Kafka，可以设置kafka的auto.offset.reset为largest的方式，使Job失败后会自动恢复Job的执行。

（2）任务推测所导致的重复输出。设置spark.speculation为关闭状态（因为任务推测其实非常消耗性能，所以关闭后可以显著提高Spark Streaming的处理性能）。

需要强调的是，在应用程序代码中，通过对DStream及其子类的transform和foreachRDD编写的代码进行逻辑控制，可以实现数据不重复消费或输出不重复。这两个方法可以做任意想象得到的控制操作。

以foreachRDD为例，可以写出以下代码：

```
dstream.foreachRDD { (rdd, time) =>
  rdd.foreachPartition { partitionIterator =>
    val partitionId = TaskContext.get.partitionId()
    val uniqueId = generateUniqueId(time.milliseconds, partitionId)
    // 使用uniqueId作为事务的唯一标识
    //基于uniqueId实现partitionIterator所指向数据的原子事务提交
    ...
  }
}
```

也就是说，针对每个Partition的数据，产生一个uniqueId，只有这个partition的所有数据被完全消费，则算成功，否则算失败，要回滚。下次重复执行这个uniqueId 时，如果是已经被执行成功过的，则跳过。这样，就能保证数据输出的exactly-once语义。

3.8 消费速率的动态控制

Spark是粗粒度的，即在默认情况下会预先分配好资源，再进行计算。好处是资源提前分配好，有计算任务时就直接使用计算资源，不用再考虑资源分配。不好的地方是，高峰值和低峰值时需要的资源是不一样的。资源如果是针对高峰值情况下考虑的，那么势必在低峰值情况下会有大量的资源浪费。

Spark Core中有资源动态分配的方式，但仍处于公开测试阶段。具体使用方法是：用spark.dynamicAllocation.enabled项进行是否动态分配的设置。

实际生产环境下，动态资源分配还是要自己做好定制。

可以考虑改变执行周期（intervalMillis）来动态调整。在一个batch duration中要对数据分片，可以算一下已拥有的闲置的core，如果不够，则可以申请增加Executor，从而把任务分配到新增的Executor。

也可以考虑针对上一个batch duration的资源需求情况，因为峰值出现时，往往会延续在多个连续的batch duration中。考量上一个batch duration的情况，用某种算法来动态调整后续的batch duration的资源。修改Spark Streaming可以设计StreamingContext的新子类。已有涉及这一方法的论文，有兴趣的读者可以自己去研究。

目前来说，Spark Streaming的动态调整仍待完善。即使在batch duration内刚做了调整，也有可能在本batch duration中马上就变得不合适。

除了动态资源分配，还可以通过消费速率的动态控制来解决资源问题，这就是所谓的反压（back pressure）机制。

这里的反压机制利用的是令牌桶机制。

先解释一下令牌桶机制的原理。大小固定的令牌桶可自行以指定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。当进行某操作需要令牌时，会从令牌桶中取出相应的令牌数，如果获取到则继续操作，否则阻塞。用完之后不用放回。

上面提到的某操作，在反压机制中是将接收到的数据放入Buffer。Streaming 数据流被Receiver接收后，按行解析后存入Iterator中。然后逐个存入Buffer，在存入Buffer时会先获取令牌，如果没有令牌存在，则阻塞；如果获取到则将数据存入Buffer，然后等后续生成Block操作。反压机制的原理如图3-18所示。

Spark Streaming中已经有动态控制速度流速度的方式，相应的配置项是spark.streaming.backpressure.enabled。默认值是false，建议设为true。

Spark Streaming提供了RateController，继承关系是RateController→StreamingListener→AsynchronousListenerBus→ListenerBus。

RateController的子类有ReceiverRateController、DirectKafkaRateController，分别定义

在ReceiverInputDStream、DirectKafkaInputDStream中。

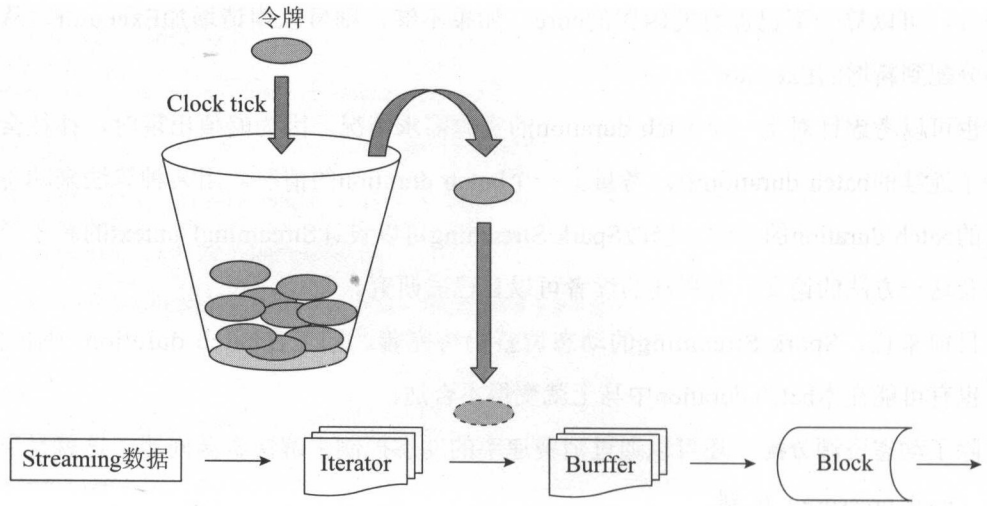


图3-18　反压机制的原理图

RateController子类一般也定义在InputDStream子类中。这应该可以理解，因为InputDStream与源数据有直接联系，而RateController是用来控制源数据的接收速率。

下面以InputDStream子类ReceiverInputDStream为例进行分析。

ReceiverInputDStream有一个成员rateController。

源码3-122　ReceiverInputDStream.rateController

```
override protected[streaming] val rateController: Option[RateController] = {
  if (RateController.isBackPressureEnabled(ssc.conf)) {
    Some(new ReceiverRateController(id, RateEstimator.create(ssc.conf, ssc.graph.batchDuration)))
  } else {
    None
  }
}
```

其中的RateController.isBackPressureEnabled获得是否允许反压机制的配置值。

源码3-123　RateController.isBackPressureEnabled

```
object RateController {
  def isBackPressureEnabled(conf: SparkConf): Boolean =
    conf.getBoolean("spark.streaming.backpressure.enabled", false)
}
```

如果允许反压机制，即spark.streaming.backpressure.enabled被设置为true，那么ReceiverInputDStream中的成员rateController被赋予新的ReceiverRateController对象；否则为None。

生成ReceiverRateController对象时会调用RateEstimator.create。

源码3-124　RateEstimator.create

```
/**
 * Return a new RateEstimator based on the value of `spark.streaming.RateEstimator`.
 *
 * The only known estimator right now is `pid`.
 *
 * @return An instance of RateEstimator
 * @throws IllegalArgumentException if there is a configured RateEstimator that doesn't match any
 *         known estimators.
 */
def create(conf: SparkConf, batchInterval: Duration): RateEstimator =
  conf.get("spark.streaming.backpressure.rateEstimator", "pid") match {
```

```
    case "pid" =>
      val proportional = conf.getDouble("spark.streaming.backpressure.pid.proportional", 1.0)
      val integral = conf.getDouble("spark.streaming.backpressure.pid.integral", 0.2)
      val derived = conf.getDouble("spark.streaming.backpressure.pid.derived", 0.0)
      val minRate = conf.getDouble("spark.streaming.backpressure.pid.minRate", 100)
      new PIDRateEstimator(batchInterval.milliseconds, proportional, integral, derived, minRate)

    case estimator =>
      throw new IllegalArgumentException(s"Unkown rate estimator: $estimator")
}
```

目前spark.streaming.backpressure.rateEstimator配置只能是pid。另外还有4个反压的可配置项。

RateEstimator用于评估InputDStream消费数据的能力。根据消费数据的能力来调整接收数据的速率。RateEstimator.create给出了反压机制。这显然要比简单限制接收速率要好一些。

BlockGenerator是Spark Streaming的RateLimiter的子类。

有必要先剖析一下RateLimiter。

源码3-125 RateLimiter

```
private[receiver] abstract class RateLimiter(conf: SparkConf) extends Logging {


  // treated as an upper limit
  private val maxRateLimit = conf.getLong("spark.streaming.receiver.maxRate", Long.MaxValue)
  private lazy val rateLimiter = GuavaRateLimiter.create(maxRateLimit.toDouble)
```

```
  def waitToPush() {
    rateLimiter.acquire()
  }

  /**
   * Return the current rate limit. If no limit has been set so far, it returns {{{Long.MaxValue}}}.
   */
  def getCurrentLimit: Long = rateLimiter.getRate.toLong

  /**
   * Set the rate limit to `newRate`. The new rate will not exceed the maximum rate configured by
   * {{{spark.streaming.receiver.maxRate}}}, even if `newRate` is higher than that.
   *
   * @param newRate A new rate in events per second. It has no effect if it's 0 or negative.
   */
  private[receiver] def updateRate(newRate: Long): Unit =
    if (newRate > 0) {
      if (maxRateLimit > 0) {
        rateLimiter.setRate(newRate.min(maxRateLimit))
      } else {
        rateLimiter.setRate(newRate)
      }
    }
}
```

RateLimiter中有一个成员rateLimiter，类型是Google Guava的限流工具类RateLimiter（看本节后面的备注）。

动态控制消费速度的过程，就是BlockGenerator利用waitToPush方法来获得限速许可，然后不断利用updateRate方法来修改速率。

首先看waitToPush是在哪里被调用。获得限速许可，离不开Receiver的配合。下面结合具体的Receiver子类来剖析，以ReliableKafkaReceiver为例。

ReliableKafkaReceiver通过内部类MessageHandler来操作Kafka信息。

源码3-126 ReliableKafkaReceiver.MessageHandler

```
/** Class to handle received Kafka message. */
private final class MessageHandler(stream: KafkaStream[K, V]) extends Runnable {
  override def run(): Unit = {
    while (!isStopped) {
      try {
        val streamIterator = stream.iterator()
        while (streamIterator.hasNext) {
          storeMessageAndMetadata(streamIterator.next)
        }
      } catch {
        case e: Exception =>
          reportError("Error handling message", e)
      }
    }
  }
}
```

MessageHandler是Runnable类，运行时调用了storeMessageAndMetadata。

源码3-127　ReliableKafkaReceiver.storeMessageAndMetadata

```
/** Store a Kafka message and the associated metadata as a tuple. */
private def storeMessageAndMetadata(
    msgAndMetadata: MessageAndMetadata[K, V]): Unit = {
  val topicAndPartition = TopicAndPartition(msgAndMetadata.topic, msgAndMetadata.partition)
  val data = (msgAndMetadata.key, msgAndMetadata.message)
  val metadata = (topicAndPartition, msgAndMetadata.offset)
  blockGenerator.addDataWithCallback(data, metadata)
}
```

源码3-128　BlockGenerator.addDataWithCallback

```
/**
 * Push a single data item into the buffer. After buffering the data, the
 * `BlockGeneratorListener.onAddData` callback will be called.
 */
def addDataWithCallback(data: Any, metadata: Any): Unit = {
  if (state == Active) {
    waitToPush()
    synchronized {
      if (state == Active) {
        currentBuffer += data
        listener.onAddData(data, metadata)
      } else {
```

```
        throw new SparkException(
          "Cannot add data as BlockGenerator has not been started or has been stopped")
      }
    }
  } else {
    throw new SparkException(
      "Cannot add data as BlockGenerator has not been started or has been stopped")
  }
}
```

addDataWithCallback调用了BlockGenerator. waitToPush。

源码3-129 BlockGenerator.waitToPush

```
def waitToPush() {
  rateLimiter.acquire()
}
```

代码rateLimiter.acquire是从Google Guava的RateLimiter获取一个许可，才能进行后续运作。本节最后对RateLimiter有专门说明。

事实上，无论哪种Receiver获取的数据，在调用BlockGenarator的加数据的方法（addData、addDataWithCallback、addMultipleDataWithCallback中某一个）时，都要调用waitToPush来获取许可，才能把数据加入到缓存；没获得许可就会阻塞，这就能达到限速效果。

限速原理清楚了。接下来看BlockGenerator如何利用RateLimiter的updateRate方法设置新的速率。显然需要在batch最后的Job完成时才能做速率评估，然后再做速率调整。所以速率控制流程是从接收JobCompleted消息后开始的。

先给出消费速率动态控制的流程，如图3-19所示。

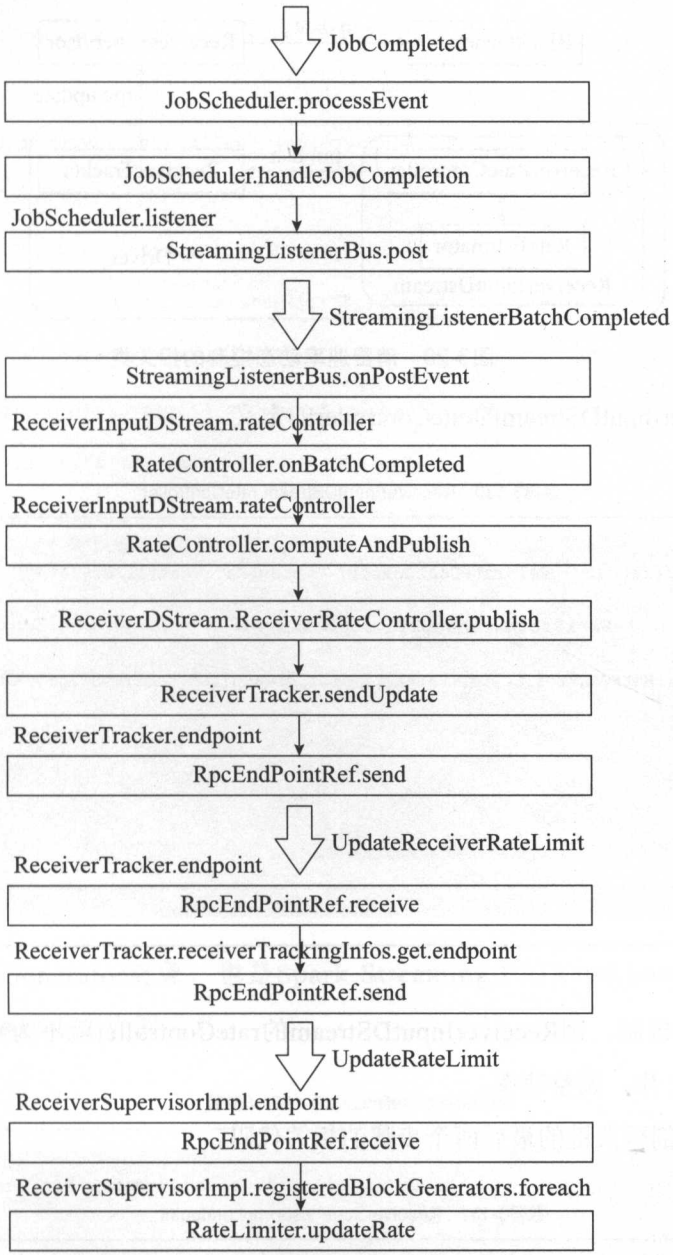


图3-19　消费速率动态控制的调用流程

图3-20体现了消费速率动态控制的相关类及其关系以及控制流程。

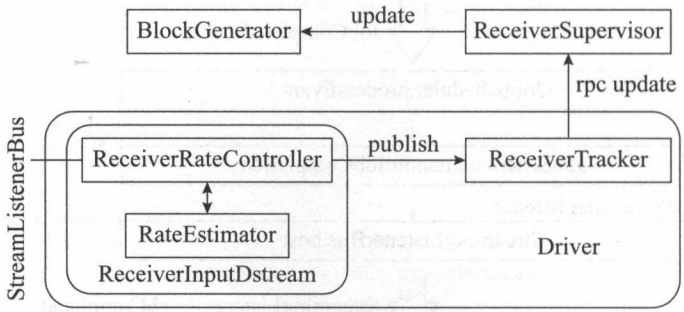


图3-20 消费速率动态控制的相关类

先看看ReceiverInputDStream的rateController的定义。

源码3-130 ReceiverInputDStream.rateController

```
override protected[streaming] val rateController: Option[RateController] = {
  if (RateController.isBackPressureEnabled(ssc.conf)) {
    Some(new ReceiverRateController(id, RateEstimator.create(ssc.conf, ssc.graph.batchDuration)))
  } else {
    None
  }
}
```

如果允许反压机制，则ReceiverInputDStream的rateController就不为None，才保证了rateController能够工作，调整速率。

流程较长，只剖析关键的最后两个步骤的相关代码。

源码3-131 ReceiverSupervisorImpl.endpoint

```
private val endpoint = env.rpcEnv.setupEndpoint(
```

```
  "Receiver-" + streamId + "-" + System.currentTimeMillis(), new ThreadSafeRpcEndpoint {
    override val rpcEnv: RpcEnv = env.rpcEnv

    override def receive: PartialFunction[Any, Unit] = {
      case StopReceiver =>
        logInfo("Received stop signal")
        ReceiverSupervisorImpl.this.stop("Stopped by driver", None)
      case CleanupOldBlocks(threshTime) =>
        logDebug("Received delete old batch signal")
        cleanupOldBlocks(threshTime)
      case UpdateRateLimit(eps) =>
        logInfo(s"Received a new rate limit: $eps.")
        registeredBlockGenerators.foreach { bg =>
          bg.updateRate(eps)
        }
    }
})
```

bg是BlockGenerator对象，也是Spark Streaming中的RateLimiter子类对象，可用RateLimiter.updateRate来设置速率。

源码3-132　RateLimiter.updateRate

```
private[receiver] def updateRate(newRate: Long): Unit =
  if (newRate > 0) {
    if (maxRateLimit > 0) {
```

```
        rateLimiter.setRate(newRate.min(maxRateLimit))
      } else {
        rateLimiter.setRate(newRate)
      }
    }
```

最后是利用Google Guava的RateLimiter来修改速率。

如果maxRateLimit也有值（即设置了spark.streaming.receiver.maxRate值），则取newRate和maxRateLimit中间的最小值。

spark.streaming.receiver.maxRate控制了最大的接收速率，但有浪费资源的可能，配置最大速率不是太好的事情。

最后，对Google Guava的限流工具类RateLimiter做一些简要介绍。

从概念上来讲，速率限制器RateLimiter会在可配置的速率下分配许可证。如果必要的话，每个acquire会阻塞当前线程直到许可证可用后获取该许可证。一旦获取许可证，就不需要再释放许可证。

RateLimiter使用的是一种叫令牌桶的流控算法，RateLimiter会按照一定的频率往令牌桶里扔令牌，线程拿到令牌才能执行，比如你希望自己的应用程序QPS（每秒查询率）不要超过1000，那么RateLimiter设置1000的速率后，就会每秒往桶里扔1000个令牌。桶中可以保存的最大令牌数永远不会超过桶的大小。当进行某操作需要限速，要获得令牌时，会从令牌桶中取出相应的令牌数，如果获取了令牌，则表示通过了许可，可以继续后续操作，否则阻塞。令牌用过之后不用放回。

RateLimiter的主要方法如表3-1所示。

表3-1 RateLimiter的主要方法

修饰符和类型	方法和描述
static RateLimiter	**create**(double permitsPerSecond)：根据指定的稳定吞吐率创建RateLimiter，这里的吞吐率是指每秒多少许可数（通常是指QPS，即每秒多少查询）

续表

修饰符和类型	方法和描述
double	**acquire**()：从RateLimiter索取一个许可，该方法会被阻塞直到获取到请求
void	**setRate**(double permitsPerSecond)：更新RateLimite的稳定速率，参数permitsPerSecond由构造RateLimiter的工厂方法提供

3.9 状态操作

Spark Streaming是按Batch Duration来划分Job的，但有时需要对某个时刻开始的某个指标进行持续跟踪和计算，并更新该指标（比如当天网页的点击次数、本月的最新销售额等）。这需要根据此前的状态结果和新Batch Duration的数据计算出新的状态结果。

状态操作updateStateByKey和mapWithState都是针对key-value（键值对）类型的数据进行操作，而RDD类本身并不对key-value类型的数据进行操作，并不存在于DStream中，所以要借助隐式转换。隐式转换一般放在伴生对象区域。DStream的伴生对象中，有一个隐式函数toPairDStreamFunctions。

源码3-133 DStream.toPairDStreamFunctions

```
implicit def toPairDStreamFunctions[K, V](stream: DStream[(K, V)])
                          (implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K]= null):
PairDStreamFunctions[K, V] = {
  new PairDStreamFunctions[K, V](stream)
}
```

以上代码生成了PairDStreamFunctions对象。也就是把DStream[(K, V)]对象隐式转换为PairDStreamFunctions对象。既然DStream类中没有updateStateByKey、mapWithState，那就看看PairDStreamFunctions类里有没有。

PairDStreamFunctions类中确实有updateStateByKey、mapWithState这些功能，如图3-21、图3-22所示。其中updateStateByKey有多个重载方法。

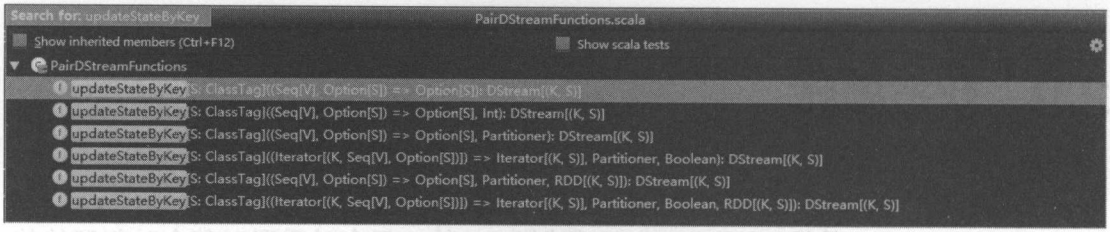


图3-21　PairDStreamFunctions类中的updateStateByKey

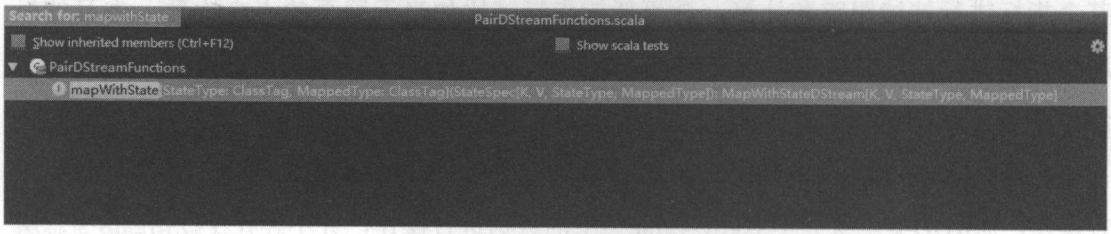


图3-22　PairDStreamFunctions类中的mapWithState

注意，在应用程序中使用状态操作，必须设置Checkpoint。

现在分析一下这些方法的性能情况。

先分析updateStateByKey，下面给出updateStateByKey的应用实例。

源码3-134　UpdateStateByKeyDemo

```
object UpdateStateByKeyDemo {

  def main(args: Array[String]) {

    val conf = new SparkConf().setAppName("UpdateStateByKeyDemo")

    val ssc = new StreamingContext(conf,Seconds(20))

    // 如果使用updateStateByKey方法，则必须设置Checkpoint
```

```
    ssc.checkpoint("/checkpoint/")
    val socketLines = ssc.socketTextStream("localhost",9999)

    socketLines.flatMap(_.split(",")).map(word=>(word,1))
      .updateStateByKey( (currValues:Seq[Int],preValue:Option[Int]) =>{
            // 目前值之和
            val currValue = currValues.sum
            // 目前值之和加上历史值
            Some(currValue + preValue.getOrElse(0))
    }).print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()

  }
}
```

上面的例子是用updateStateByKey把目前Batch Interval内各单词的个数之和加上其历史总数，得到新的总和。

下面分析源码。先看updateStateByKey。

源码3-135　PairDStreamFunctions.updateStateByKey

```
/**
 * Return a new "state" DStream where the state for each key is updated by applying
```

```
 * the given function on the previous state of the key and the new values of each key.
 * Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
 * @param updateFunc  State update function. If `this` function returns None, then
 *                    corresponding state key-value pair will be eliminated.
 * @tparam S          State type
 */
def updateStateByKey[S: ClassTag](
    updateFunc: (Seq[V], Option[S]) => Option[S]
  ): DStream[(K, S)] = ssc.withScope {
  updateStateByKey(updateFunc, defaultPartitioner())
}
```

updateStateByKey返回的是DStream类型。

根据updateFunc这个函数来更新状态。其中，参数Seq[V]是本次的数据类型，Option[S]是前次计算结果类型，本次计算结果类型是Option[S]。

计算肯定需要Partitioner。因为Hash效率高且不做排序，故默认Partitioner是HashPartitoner。

源码3-136 PairDStreamFunction.defaultPartitioner

```
private[streaming] def defaultPartitioner(numPartitions: Int = self.ssc.sc.defaultParallelism) = {
  new HashPartitioner(numPartitions)
}
```

其他updateStateByKey最终都会调用返回值类型为StateDStream的updateStateByKey。

源码3-137　PairDStreamFunctions.updateStateByKey

```
/**
 * Return a new "state" DStream where the state for each key is updated by applying
 * the given function on the previous state of the key and the new values of each key.
 * org.apache.spark.Partitioner is used to control the partitioning of each RDD.
 * @param updateFunc  State update function. Note, that this function may generate a different
 *                    tuple with a different key than the input key. Therefore keys may be removed
 *                    or added in this way. It is up to the developer to decide whether to
 *                    remember the partitioner despite the key being changed.
 * @param partitioner Partitioner for controlling the partitioning of each RDD in the new
 *                    DStream
 * @param rememberPartitioner
 *                    Whether to remember the paritioner object in the generated RDDs.
 * @tparam S          State type
 */
def updateStateByKey[S: ClassTag](
    updateFunc: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)],
    partitioner: Partitioner,
    rememberPartitioner: Boolean
  ): DStream[(K, S)] = ssc.withScope {
   new StateDStream(self, ssc.sc.clean(updateFunc), partitioner, rememberPartitioner, None)
}
```

接着看看StateDStream类。

源码3-138 StateDStream片段

```
class StateDStream[K: ClassTag, V: ClassTag, S: ClassTag](
      parent: DStream[(K, V)],
      updateFunc: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)],
      partitioner: Partitioner,
      preservePartitioning: Boolean,
      initialRDD : Option[RDD[(K, S)]]
  ) extends DStream[(K, S)](parent.ssc) {

  super.persist(StorageLevel.MEMORY_ONLY_SER)
...
```

以上代码是只基于内存的序列化。

StateDStream作为DStream子类，要覆写compute方法。

源码3-139 StateDStream.compute

```
override def compute(validTime: Time): Option[RDD[(K, S)]] = {

  // Try to get the previous state RDD
  getOrCompute(validTime - slideDuration) match {

    case Some(prevStateRDD) => {    // If previous state RDD exists

      // Try to get the parent RDD
      parent.getOrCompute(validTime) match {
```

```
    case Some(parentRDD) => {   // If parent RDD exists, then compute as usual
      computeUsingPreviousRDD (parentRDD, prevStateRDD)
    }
    case None => {     // If parent RDD does not exist

      // Re-apply the update function to the old state RDD
      val updateFuncLocal = updateFunc
      val finalFunc = (iterator: Iterator[(K, S)]) => {
        val i = iterator.map(t => (t._1, Seq[V](), Option(t._2)))
        updateFuncLocal(i)
      }
      val stateRDD = prevStateRDD.mapPartitions(finalFunc, preservePartitioning)
      Some(stateRDD)
    }
  }
}

case None => {    // If previous session RDD does not exist (first input data)

  // Try to get the parent RDD
  parent.getOrCompute(validTime) match {
    case Some(parentRDD) => {   // If parent RDD exists, then compute as usual
      initialRDD match {
        case None => {
          // Define the function for the mapPartition operation on grouped RDD;
          // first map the grouped tuple to tuples of required type,
```

```
            // and then apply the update function
            val updateFuncLocal = updateFunc
            val finalFunc = (iterator : Iterator[(K, Iterable[V])]) => {
              updateFuncLocal (iterator.map (tuple => (tuple._1, tuple._2.toSeq, None)))
            }

            val groupedRDD = parentRDD.groupByKey (partitioner)
            val sessionRDD = groupedRDD.mapPartitions (finalFunc, preservePartitioning)
            // logDebug("Generating state RDD for time "+ validTime + "(first)")
            Some (sessionRDD)
          }
          case Some (initialStateRDD) => {
            computeUsingPreviousRDD(parentRDD, initialStateRDD)
          }
        }
        case None => { // If parent RDD does not exist, then nothing to do!
          // logDebug("Not generating state RDD (no previous state, no parent)")
          None
        }
      }
    }
  }
}
```

compute会用到computeUsingPreviousRDD方法。

源码3-140　StateDStream.computeUsingPreviousRDD

```
private [this] def computeUsingPreviousRDD (
  parentRDD : RDD[(K, V)], prevStateRDD : RDD[(K, S)]) = {
  // Define the function for the mapPartition operation on cogrouped RDD;
  // first map the cogrouped tuple to tuples of required type,
  // and then apply the update function
  val updateFuncLocal = updateFunc
  val finalFunc = (iterator: Iterator[(K, (Iterable[V], Iterable[S]))]) => {
    val i = iterator.map(t => {
      val itr = t._2._2.iterator
      val headOption = if (itr.hasNext) Some(itr.next()) else None
      (t._1, t._2._1.toSeq, headOption)
    })
    updateFuncLocal(i)
  }
  val cogroupedRDD = parentRDD.cogroup(prevStateRDD, partitioner)
  val stateRDD = cogroupedRDD.mapPartitions(finalFunc, preservePartitioning)
  Some(stateRDD)
}
```

由于cogroup会对所有数据进行扫描，再按key进行分组，所以性能上会有问题。特别是随着时间的推移，这样的计算到后面会越来越慢。

所以，对于数据量大的计算，不建议使用updateStateByKey。

虽然使用mapWithState感觉效果尚可，但源码中表明，mapWithState仍在试验状态。

Spark中提供了例程StatefulNetworkWordCount。

源码3-141 StatefulNetworkWordCount

```
/**
 * Counts words cumulatively in UTF8 encoded, '\n' delimited text received from the network every
 * second starting with initial value of word count.
 * Usage: StatefulNetworkWordCount <hostname> <port>
 *   <hostname> and <port> describe the TCP server that Spark Streaming would connect to receive
 *   data.
 *
 * To run this on your local machine, you need to first run a Netcat server
 *     `$ nc -lk 9999`
 * and then run the example
 *     `$ bin/run-example
 *       org.apache.spark.examples.streaming.StatefulNetworkWordCount localhost 9999`
 */
object StatefulNetworkWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: StatefulNetworkWordCount <hostname> <port>")
      System.exit(1)
    }

    StreamingExamples.setStreamingLogLevels()

    val sparkConf = new SparkConf().setAppName("StatefulNetworkWordCount")
    // Create the context with a 1 second batch size
```

```
val ssc = new StreamingContext(sparkConf, Seconds(1))
ssc.checkpoint(".")

// Initial state RDD for mapWithState operation
val initialRDD = ssc.sparkContext.parallelize(List(("hello", 1), ("world", 1)))

// Create a ReceiverInputDStream on target ip:port and count the
// words in input stream of \n delimited test (eg. generated by 'nc')
val lines = ssc.socketTextStream(args(0), args(1).toInt)
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

// Update the cumulative count using mapWithState
// This will give a DStream made of state (which is the cumulative
   count of the words)
val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
  output
}

val stateDstream = wordDstream.mapWithState(
  StateSpec.function(mappingFunc).initialState(initialRDD))
stateDstream.print()
ssc.start()
```

```
    ssc.awaitTermination()
  }
}
```

下面看mapWithState方法。

源码3-142 PairDStreamFunctions.mapWithState

```
/**
 * :: Experimental ::
 * Return a [[MapWithStateDStream]] by applying a function to every key-value element of
 * `this` stream, while maintaining some state data for each unique key. The mapping function
 * and other specification (e.g. partitioners, timeouts, initial state data, etc.) of this
 * transformation can be specified using [[StateSpec]] class. The state data is accessible in
 * as a parameter of type [[State]] in the mapping function.
 *
 * Example of using `mapWithState`:
 * {{{
 *    // A mapping function that maintains an integer state and return a String
 *    def mappingFunction(key: String, value: Option[Int], state: State[Int]): Option[String] = {
 *      // Use state.exists(), state.get(), state.update() and state. remove()
 *      // to manage state, and return the necessary string
 *    }
 *
 *    val spec = StateSpec.function(mappingFunction).numPartitions(10)
 *
```

```
 *    val mapWithStateDStream = keyValueDStream.mapWithState[StateType, MappedType](spec)
 * }}}
 *
 * @param spec          Specification of this transformation
 * @tparam StateType    Class type of the state data
 * @tparam MappedType   Class type of the mapped data
 */
@Experimental
def mapWithState[StateType: ClassTag, MappedType: ClassTag](
    spec: StateSpec[K, V, StateType, MappedType]
  ): MapWithStateDStream[K, V, StateType, MappedType] = {
  new MapWithStateDStreamImpl[K, V, StateType, MappedType](
    self,
    spec.asInstanceOf[StateSpecImpl[K, V, StateType, MappedType]]
  )
}
```

注释中给出了一个mapWithState使用实例。先要定义一个mappingFunction。在mappingFunction的参数中，State类型的state是历史数据，相当于一个内存数据表；key指明是对state中的哪个键进行操作；value指明键值。

StateSpec类型的参数中封装了mapping功能和转换的相应配置（例如partitioner、超时设定、初始状态数据等）。

mapWithState返回的是MapWithStateDStream类型。

来看看State类，其中的注释有例子参考。

源码3-143 State片段

```
/**
 * :: Experimental ::
 * Abstract class for getting and updating the state in mapping function used in the `mapWithState`
 * operation of a [[org.apache.spark.streaming.dstream.PairDStreamFunctions pair DStream]] (Scala)
 * or a [[org.apache.spark.streaming.api.java.JavaPairDStream JavaPair DStream]] (Java).
 *
 * Scala example of using `State`:
 * {{{
 *    // A mapping function that maintains an integer state and returns a String
 *    def mappingFunction(key: String, value: Option[Int], state: State[Int]): Option[String] = {
 *      // Check if state exists
 *      if (state.exists) {
 *        val existingState = state.get          // Get the existing state
 *        val shouldRemove = ...                 // Decide whether to remove the state
 *        if (shouldRemove) {
 *          state.remove()                       // Remove the state
 *        } else {
 *          val newState = ...
 *          state.update(newState)               // Set the new state
 *        }
 *      } else {
 *        val initialState = ...
 *        state.update(initialState)             // Set the initial state
 *      }
```

```
 *       ...                                          // return something
 *     }
 *
 * }}}
 *
...
sealed abstract class State[S] {
...
```

State中有exists、get、update、remove、isTimingOut等需要在子类中覆写的方法。

State中还有一个内部实现类StateImpl。

源码3-144　StateImpl

```
/** Internal implementation of the [[State]] interface */
private[streaming] class StateImpl[S] extends State[S] {

  private var state: S = null.asInstanceOf[S]
  private var defined: Boolean = false
  private var timingOut: Boolean = false
  private var updated: Boolean = false
  private var removed: Boolean = false

  // ========= Public API =========
  override def exists(): Boolean = {
```

```
    defined
  }
...
```

StateImpl有一些状态变量，并且覆写了State中的方法。

回去再看PairDStreamFunctions中的其他mapWithState方法。

源码3-145 PairDStreamFunctions.mapWithState

```
@Experimental
def mapWithState[StateType: ClassTag, MappedType: ClassTag](
    spec: StateSpec[K, V, StateType, MappedType]
  ): MapWithStateDStream[K, V, StateType, MappedType] = {
  new MapWithStateDStreamImpl[K, V, StateType, MappedType](
    self,
    spec.asInstanceOf[StateSpecImpl[K, V, StateType, MappedType]]
  )
}
```

先看看StateSpecImpl。StateSpecImpl是StateSpec类中的case class。

源码3-146 StateSpecImpl片段

```
/** Internal implementation of [[org.apache.spark.streaming.StateSpec]] interface. */
private[streaming]
case class StateSpecImpl[K, V, S, T](
```

```
function: (Time, K, Option[V], State[S]) => Option[T]) extends StateSpec[K, V, S, T] {
...
```

其参数是一个函数。

源码3-147　StateSpecImpl的参数

```
require(function != null)

@volatile private var partitioner: Partitioner = null
@volatile private var initialStateRDD: RDD[(K, S)] = null
@volatile private var timeoutInterval: Duration = null
...
  // ================= Private Methods =================

private[streaming] def getFunction(): (Time, K, Option[V], State[S]) => Option[T] = function

private[streaming] def getInitialStateRDD(): Option[RDD[(K, S)]] = Option(initialStateRDD)

private[streaming] def getPartitioner(): Option[Partitioner] = Option(partitioner)

private[streaming] def getTimeoutInterval(): Option[Duration] = Option(timeoutInterval)
```

上面的代码中有一些私有变量及其变量的获取方法，特别是有一个函数的获取方法。

再回到源码mapWithState。mapWithState实际返回的是MapWithStateDStreamImpl

对象。

MapWithStateDStreamImpl被定义在MapWithStateDStream中。

MapWithStateDStream类包含4部分：

（1）抽象类MapWithStateDStream。

（2）MapWithStateDStream子类MapWithStateDStreamImpl。

（3）DStream子类InternalMapWithStateDStream。

（4）InternalMapWithStateDStream的伴生对象。

来看看MapWithStateDStreamImpl。

源码3-148 MapWithStateDStreamImpl代码片段

```
/** Internal implementation of the [[MapWithStateDStream]] */
private[streaming] class MapWithStateDStreamImpl[
    KeyType: ClassTag, ValueType: ClassTag, StateType: ClassTag, MappedType: ClassTag](
    dataStream: DStream[(KeyType, ValueType)],
    spec: StateSpecImpl[KeyType, ValueType, StateType, MappedType])
  extends MapWithStateDStream[KeyType, ValueType, StateType, MappedType](dataStream.context) {

  private val internalStream =
    new InternalMapWithStateDStream[KeyType, ValueType, StateType, MappedType](dataStream, spec)

  override def slideDuration: Duration = internalStream.slideDuration

  override def dependencies: List[DStream[_]] = List(internalStream)

  override def compute(validTime: Time): Option[RDD[MappedType]] = {
```

```
    internalStream.getOrCompute(validTime).map { _.flatMap[MappedType] { _.mappedData } }
}
```

其中生成了一个DStream子类InternalMapWithStateDStream的对象。

源码3-149 InternalMapWithStateDStream.compute

```
/** Method that generates a RDD for the given time */
override def compute(validTime: Time): Option[RDD[MapWithStateRDDRecord[K, S, E]]] = {
  // Get the previous state or create a new empty state RDD
  val prevStateRDD = getOrCompute(validTime - slideDuration) match {
    case Some(rdd) =>
      if (rdd.partitioner != Some(partitioner)) {
        // If the RDD is not partitioned the right way, let us repartition it using the
        // partition index as the key. This is to ensure that state RDD is always partitioned
        // before creating another state RDD using it
        MapWithStateRDD.createFromRDD[K, V, S, E](
          rdd.flatMap { _.stateMap.getAll() }, partitioner, validTime)
      } else {
        rdd
      }
    case None =>
      MapWithStateRDD.createFromPairRDD[K, V, S, E](
        spec.getInitialStateRDD().getOrElse(new EmptyRDD[(K, S)](ssc.sparkContext)),
        partitioner,
        validTime
```

```
      )
    }

    // Compute the new state RDD with previous state RDD and partitioned data RDD
    // Even if there is no data RDD, use an empty one to create a new state RDD
    val dataRDD = parent.getOrCompute(validTime).getOrElse {
      context.sparkContext.emptyRDD[(K, V)]
    }
    val partitionedDataRDD = dataRDD.partitionBy(partitioner)
    val timeoutThresholdTime = spec.getTimeoutInterval().map { interval =>
      (validTime - interval).milliseconds
    }
    Some(new MapWithStateRDD(
      prevStateRDD, partitionedDataRDD, mappingFunction, validTime, timeoutThresholdTime))
  }
```

其中生成了RDD子类MapWithStateRDD的对象。

源码3-150 MapWithStateRDD

```
private[streaming] class MapWithStateRDD[K: ClassTag, V: ClassTag, S: ClassTag, E: ClassTag](
    private var prevStateRDD: RDD[MapWithStateRDDRecord[K, S, E]],
    private var partitionedDataRDD: RDD[(K, V)],
    mappingFunction: (Time, K, Option[V], State[S]) => Option[E],
    batchTime: Time,
    timeoutThresholdTime: Option[Long]
```

```
  ) extends RDD[MapWithStateRDDRecord[K, S, E]](
    partitionedDataRDD.sparkContext,
    List(
      new OneToOneDependency[MapWithStateRDDRecord[K, S, E]](prevStateRDD),
      new OneToOneDependency(partitionedDataRDD))
  ) {
...
```

MapWithStateRDD中每个分区由一个类型为MapWithStateRDDRecord的记录所代表。

源码3-151　MapWithStateRDD.compute

```
override def compute(
    partition: Partition, context: TaskContext): Iterator[MapWithStateRDDRecord[K, S, E]] = {

  val stateRDDPartition = partition.asInstanceOf[MapWithStateRDDPartition]
  val prevStateRDDIterator = prevStateRDD.iterator(
    stateRDDPartition.previousSessionRDDPartition, context)
  val dataIterator = partitionedDataRDD.iterator(
    stateRDDPartition.partitionedDataRDDPartition, context)

  val prevRecord = if (prevStateRDDIterator.hasNext) Some(prevStateRDDIterator.next()) else None
  val newRecord = MapWithStateRDDRecord.updateRecordWithData(
    prevRecord,
    dataIterator,
    mappingFunction,
```

```
    batchTime,
    timeoutThresholdTime,
    removeTimedoutData = doFullScan     // remove timedout data only when full scan is enabled
  )
  Iterator(newRecord)
}
```

MapWithStateRDDRecord有伴生对象，定义了updateRecordWithData。

源码3-152 MapWithStateRDDRecord伴生对象

```
private[streaming] object MapWithStateRDDRecord {
  def updateRecordWithData[K: ClassTag, V: ClassTag, S: ClassTag, E: ClassTag](
    prevRecord: Option[MapWithStateRDDRecord[K, S, E]],
    dataIterator: Iterator[(K, V)],
    mappingFunction: (Time, K, Option[V], State[S]) => Option[E],
    batchTime: Time,
    timeoutThresholdTime: Option[Long],
    removeTimedoutData: Boolean
  ): MapWithStateRDDRecord[K, S, E] = {
    // Create a new state map by cloning the previous one (if it exists) or by creating an empty one
    val newStateMap = prevRecord.map { _.stateMap.copy() }. getOrElse { new EmptyStateMap[K,S]() }

    val mappedData = new ArrayBuffer[E]
    val wrappedState = new StateImpl[S]()
```

```
// Call the mapping function on each record in the data iterator, and accordingly
// update the states touched, and collect the data returned by the mapping function
// 此处是mapWithState性能较好的核心代码之所在
dataIterator.foreach { case (key, value) =>
  wrappedState.wrap(newStateMap.get(key))
  val returned = mappingFunction(batchTime, key, Some(value), wrappedState)
  if (wrappedState.isRemoved) {
    newStateMap.remove(key)
  } else if (wrappedState.isUpdated
      || (wrappedState.exists && timeoutThresholdTime.isDefined)) {
    newStateMap.put(key, wrappedState.get(), batchTime.milliseconds)
  }
  mappedData ++= returned
}

// Get the timed out state records, call the mapping function on each and collect the
// data returned
if (removeTimedoutData && timeoutThresholdTime.isDefined) {
  newStateMap.getByTime(timeoutThresholdTime.get).foreach { case (key, state, _) =>
    wrappedState.wrapTimingOutState(state)
    val returned = mappingFunction(batchTime, key, None, wrappedState)
    mappedData ++= returned
    newStateMap.remove(key)
  }
}
```

```
    MapWithStateRDDRecord(newStateMap, mappedData)
  }
```

MapWithStateRDDRecord借助了RDD的不变性，同时也借助了可变化特征，完成了高效的处理过程。

RDD不可变，但数据源可以变。

所以不可变的RDD也可用于处理变化的数据。

3.10 窗口操作

Spark Streaming应用程序中，对于DStream除了常用的转换操作、输出操作、状态操作，还有窗口操作（window operation）。窗口操作，即周期性地对过去固定长度的最近时间段内的数据进行处理。可以形象地把这个时间段看成是一个滑动的窗口，因为时间段长度固定，而操作的起始时刻是在等时后延的。窗口操作设计了两个参数：一个是滑动窗口的宽度（window duration），另一个是窗口滑动的频率（slide duration），这两个参数必须是 Batch Duration 的整数倍。

以图3-23为例。

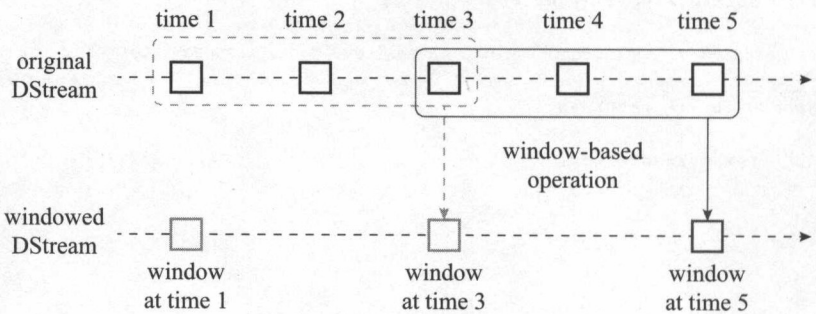


图3-23 窗口操作示例

图3-22中上半部分显示了两个滑动窗口，代表了两次窗口操作，DStream的窗口操作会生成图3-22中下半部分的WindowedDStream。

常用的窗口操作有window、reduceByWindow、reduceByKeyAndWindow、countByWindow、countByValueAndWindow等，如表3-2所示。

表3-2　常用的窗口操作

窗口操作	描　述
window(windowLength, slideInterval)	返回一个基于源DStream的窗口批次计算后得到的新的DStream
countByWindow(windowLength, slideInterval)	返回基于滑动窗口的DStream中的元素的数量
reduceByWindow(func, windowLength,slideInterval)	基于滑动窗口对源DStream中的元素进行聚合操作，得到一个新的DStream
reduceByKeyAndWindow(func, windowLength,slideInterval, [numTasks])	基于滑动窗口对键值对（K，V）类型的DStream中的值按K使用聚合函数func进行聚合操作，得到一个新的DStream
reduceByKeyAndWindow (func,invFunc,windowLength, slideInterval, [numTasks])	一个更高效的reduceByKeyAndWindow()的实现版本，先对滑动窗口中新的时间间隔内数据增量聚合并移去最早的与新增数据量的时间间隔内的数据统计量。例如，计算t+4秒这个时刻过去5秒窗口的WordCount，那么可以将t+3时刻过去5秒的统计量加上[t+3，t+4]的统计量，再减去[t−2，t−1]的统计量，这种方法可以复用中间3秒的统计量，提高统计的效率
countByValueAndWindow (windowLength,slideInterval, [numTasks])	基于滑动窗口计算源DStream中每个RDD内每个元素出现的频次并返回DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素频次。与countByValue一样，reduce任务的数量可以通过一个可选参数进行配置

以NetworkWordCount（源码3-1）为例，如果要每隔10s统计一次过去30s过来的数据，则需增加代码如下：

```
val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _, Seconds(30), Seconds(10))
```

这里paris是前面代码生成的一个MapedRDD，类似（word，1）。

和其他DStream一样，WindowedDStream的生成需要有自己的compute方法。

源码3-153 WindowedDStream.compute

```
override def compute(validTime: Time): Option[RDD[T]] = {
  val currentWindow = new Interval(validTime - windowDuration + parent.slideDuration, validTime)
  val rddsInWindow = parent.slice(currentWindow)
  val windowRDD = if (rddsInWindow.flatMap(_.partitioner).distinct.length == 1) {
    logDebug("Using partition aware union for windowing at "+ validTime)
    new PartitionerAwareUnionRDD(ssc.sc, rddsInWindow)
  } else {
    logDebug("Using normal union for windowing at " + validTime)
    new UnionRDD(ssc.sc, rddsInWindow)
  }
  Some(windowRDD)
}
```

首先生成Interval对象。两个参数的计算公式如下：

起始时刻=当前有效时刻-窗口周期时长+父DStream的滑动周期长

结束时刻=当前有效时刻

然后利用父DStream的slice方法滑动，如果滑动窗口的数量是1，则创建PartitionerAwareUnionRDD对象，否则创建UnionRDD对象。

下面分析Interval类。

源码3-154 Interval

```
class Interval(val beginTime: Time, val endTime: Time) {
  def this(beginMs: Long, endMs: Long) = this(new Time(beginMs), new Time(endMs))
...
```

WindowedDStream.compute生成的Interval对象就是一个时间段。

然后调用了DStream.slice。

源码3-155　DStream.slice

```
/**
 * Return all the RDDs defined by the Interval object (both end times included)
 */
def slice(interval: Interval): Seq[RDD[T]] = ssc.withScope {
  slice(interval.beginTime, interval.endTime)
}

/**
 * Return all the RDDs between 'fromTime' to 'toTime' (both included)
 */
def slice(fromTime: Time, toTime: Time): Seq[RDD[T]] = ssc.withScope {
  if (!isInitialized) {
    throw new SparkException(this + " has not been initialized")
  }

  val alignedToTime = if ((toTime - zeroTime).isMultipleOf(slideDuration)) {
    toTime
  } else {
    logWarning("toTime ("+ toTime + ") is not a multiple of slideDuration ("
        + slideDuration + ")")
      toTime.floor(slideDuration, zeroTime)
  }
```

```
  val alignedFromTime = if ((fromTime - zeroTime).isMultipleOf(slideDuration)) {
    fromTime
  } else {
    logWarning("fromTime ("+ fromTime + ") is not a multiple of slideDuration ("
    + slideDuration + ")")
    fromTime.floor(slideDuration, zeroTime)
  }

  logInfo("Slicing from " + fromTime + " to " + toTime +
    "(aligned to "+ alignedFromTime + "and "+ alignedToTime + ")")

  alignedFromTime.to(alignedToTime, slideDuration).flatMap(time => {
    if (time >= zeroTime) getOrCompute(time) else None
  })
}
```

上面的代码用来返回时间段的所有RDD。前面的slice调用了后面的slice。

3.11 页面展示

Spark提供了监控功能，用浏览器可访问具有样式及布局、提供丰富监控数据的页面。Spark UI架构如图3-24所示。

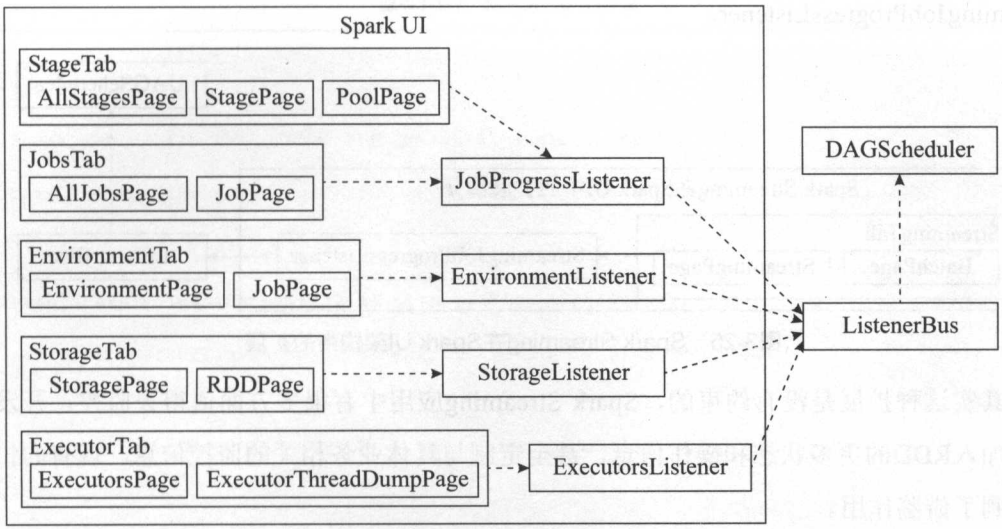


图3-24　Spark UI架构

DAGScheduler是产生各类SparkListenerEvent的主要源头，会将各种SparkListenerEvent发送到ListenerBus的事件队列中，ListenerBus通过定时器将SparkListenerEvent事件匹配到具体的SparkListener，改变SparkListener中的统计监控数据，最终由Spark UI的界面展示。

Spark中负责监控服务的Spark UI提供了这种简单、高效的页面展示方式。Spark Streaming相关监视信息的展示也是通过Spark UI来实现的。

Spark定义了一套自己的页面体系。其对象分两类：

（1）框架类。就是维护各个页面关系，和Jetty API有关联，负责管理的相关类。最基本的抽象类是WebUI。SparkUI类继承自WebUI，是中枢类，负责启动jetty，保存页面和URL Path之间的关系等。

（2）页面类。比如页面的Tab、页面渲染的内容等。最基本的抽象类是WebUITab。SparkUITab类继承自WebUITab，就是首页的标签栏。其中会有若干个WebUIPage。

Spark Streaming在以上架构的基础上进行了扩展，如图3-25所示。增加了WebUITab子类StreamingTab，增加了两个WebUIPage子类：BatchPage、StreamingPage。为实现事件侦听，增加了ListenerBus子类StreamingListenerBus和SparkListener子类

StreamingJobProgressListener。

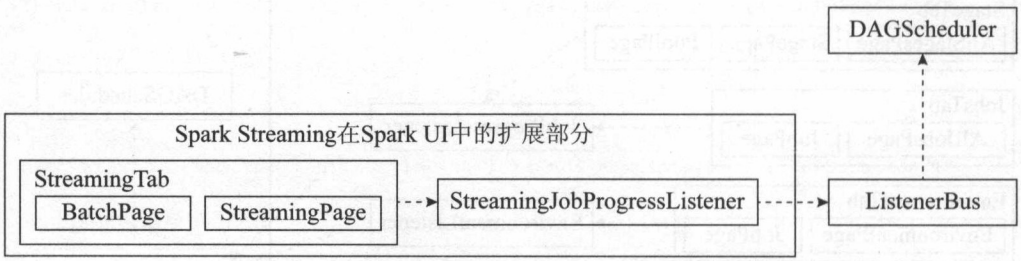


图3-25 Spark Streaming在Spark UI架构中的扩展

其实这种扩展是没有约束的，Spark Streaming应用中有很多方面值得去监控。开发者可以加入RDD的更多状态和操作信息，甚至定制与具体业务相关的监控信息。现有的扩展也起到了借鉴作用。

在StreamingContext中，初始化了StreamingJobProgressListener实例。

源码3-156 StreamingContext片段

```
...
private[streaming] val progressListener = new StreamingJobProgressListener(this)

private[streaming] val uiTab: Option[StreamingTab] =
  if (conf.getBoolean("spark.ui.enabled", true)) {
    Some(new StreamingTab(this))
  } else {
    None
  }
...
```

StreamingTab用于展示流式作业的统计信息。

源码3-157 StreamingTab片段

```
/**
 * Spark Web UI tab that shows statistics of a streaming job.
 * This assumes the given SparkContext has enabled its SparkUI.
 */
private[spark] class StreamingTab(val ssc: StreamingContext)
    extends SparkUITab(getSparkUI(ssc), "streaming") with Logging {

      private val STATIC_RESOURCE_DIR = "org/apache/spark/streaming/ui/static"

      val parent = getSparkUI(ssc)
      val listener = ssc.progressListener

      ssc.addStreamingListener(listener)
      ssc.sc.addSparkListener(listener)
      attachPage(new StreamingPage(this))
      attachPage(new BatchPage(this))
      ...

}
...
```

首先，把listener注册给streamingContext和sparkContext。其次，通过attachPage方法添加页面BatchPage、StreamingPage。

BatchPage、StreamingPage都靠render方法渲染页面。其中有页面格式化的DIV+CSS内容。

源码3-158 BatchPage.render

```
def render(request: HttpServletRequest): Seq[Node] = streamingListener.synchronized {
  val batchTime = Option(request.getParameter("id")).map(id => Time(id.toLong)).getOrElse {
    throw new IllegalArgumentException(s"Missing id parameter")
  }
  val formattedBatchTime =
    UIUtils.formatBatchTime(batchTime.milliseconds, streamingListener.batchDuration)

  val batchUIData = streamingListener.getBatchUIData(batchTime).getOrElse {
    throw new IllegalArgumentException(s"Batch $formattedBatchTime does not exist")
  }

  val formattedSchedulingDelay =
    batchUIData.schedulingDelay.map(SparkUIUtils.formatDuration).getOrElse("-")
  val formattedProcessingTime =
    batchUIData.processingDelay.map(SparkUIUtils.formatDuration).getOrElse("-")
  val formattedTotalDelay = batchUIData.totalDelay.map(SparkUIUtils.formatDuration).getOrElse("-")

  val inputMetadatas = batchUIData.streamIdToInputInfo.values.flatMap { inputInfo =>
    inputInfo.metadataDescription.map(desc => inputInfo.inputStreamId -> desc)
  }.toSeq
  val summary: NodeSeq =
    <div>
      <ul class="unstyled">
        <li>
```

```
    <strong>Batch Duration: </strong>
    {SparkUIUtils.formatDuration(streamingListener.batchDuration)}
  </li>
  <li>
    <strong>Input data size: </strong>
    {batchUIData.numRecords} records
  </li>
  <li>
    <strong>Scheduling delay: </strong>
    {formattedSchedulingDelay}
  </li>
  <li>
    <strong>Processing time: </strong>
    {formattedProcessingTime}
  </li>
  <li>
    <strong>Total delay: </strong>
    {formattedTotalDelay}
  </li>
  {
    if (inputMetadatas.nonEmpty) {
      <li>
        <strong>Input Metadata:</strong>{generateInputMetadataTable(inputMetadatas)}
      </li>
    }
  }
```

```
      </ul>
    </div>

  val content = summary ++ generateJobTable(batchUIData)

  SparkUIUtils.headerSparkPage(s"Details of batch at $formattedBatchTime", content, parent)
}
```

源码3-159 StreamingPage.render

```
/** Render the page */
def render(request: HttpServletRequest): Seq[Node] = {
  val resources = generateLoadResources()
  val basicInfo = generateBasicInfo()
  val content = resources ++
    basicInfo ++
    listener.synchronized {
      generateStatTable() ++
        generateBatchListTables()
    }
  SparkUIUtils.headerSparkPage("Streaming Statistics", content, parent, Some(5000))
}
```

两个WebUIPage的render定义了布局、页面内容，设置了相关的变量、函数等，最后都调用SparkUIUtils.headerSparkPage，来渲染出头部风格统一的页面。

接下来以Job完成为例，来分析相关的页面展示的流程，如图3-26所示。

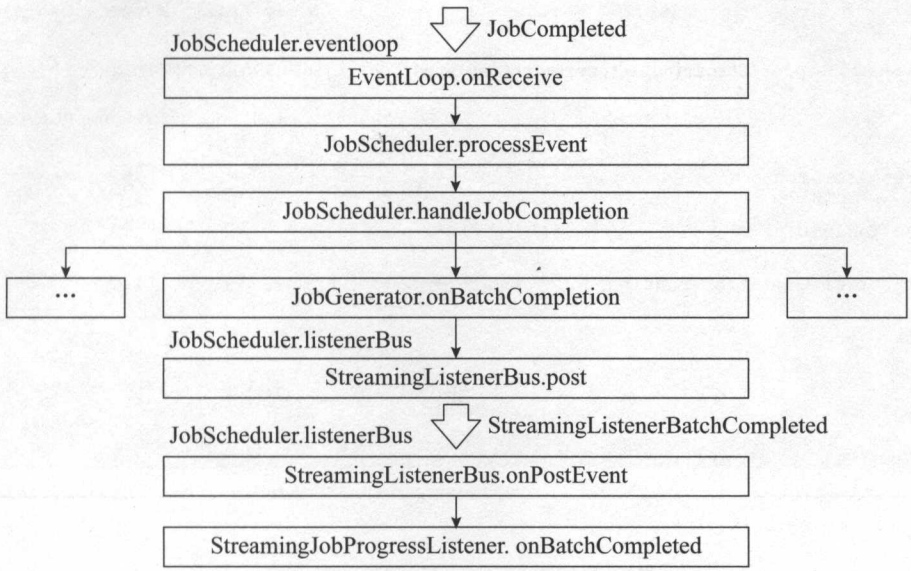


图3-26　Job完成后与StreamingPage显示相关的流程

源码3-160　JobScheduler.handleJobCompletion

```
private def handleJobCompletion(job: Job, completedTime: Long) {
  val jobSet = jobSets.get(job.time)
  jobSet.handleJobCompletion(job)
  job.setEndTime(completedTime)
  listenerBus.post(StreamingListenerOutputOperationCompleted(job.toOutputOperationInfo))
  logInfo("Finished job "+ job.id + "from job set of time "+ jobSet.time)
  if (jobSet.hasCompleted) {
    jobSets.remove(jobSet.time)
    jobGenerator.onBatchCompletion(jobSet.time)
    logInfo("Total delay: %.3f s for time %s (execution: %.3f s)".format(
      jobSet.totalDelay / 1000.0, jobSet.time.toString,
```

```
      jobSet.processingDelay / 1000.0
    ))
    listenerBus.post(StreamingListenerBatchCompleted(jobSet.toBatchInfo))
  }
  job.result match {
    case Failure(e) =>
      reportError("Error running job " + job, e)
    case _ =>
  }
}
```

源码3-161 StreamingListenerBus.onPostEvent

```
override def onPostEvent(listener: StreamingListener, event: StreamingListenerEvent): Unit = {
  event match {
    ...
    case batchCompleted: StreamingListenerBatchCompleted =>
      listener.onBatchCompleted(batchCompleted)
    ...
  }
}
```

源码3-162 StreamingJobProgressListener.onBatchCompleted

```
override def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted): Unit = {
  synchronized {
```

```
    waitingBatchUIData.remove(batchCompleted.batchInfo.batchTime)
    runningBatchUIData.remove(batchCompleted.batchInfo.batchTime)
    val batchUIData = BatchUIData(batchCompleted.batchInfo)
    completedBatchUIData.enqueue(batchUIData)
    if (completedBatchUIData.size > batchUIDataLimit) {
      val removedBatch = completedBatchUIData.dequeue()
      batchTimeToOutputOpIdSparkJobIdPair.remove(removedBatch.batchTime)
    }
    totalCompletedBatches += 1L

    totalProcessedRecords += batchUIData.numRecords
  }
}
```

StreamingJobProgressListener中的成员waitingBatchUIData、runningBatchUIData、completedBatchUIData分别在waitingBatches、runningBatches、retainedCompletedBatches函数中会被使用。

源码3-163　StreamingJobProgressListener片段

```
def waitingBatches: Seq[BatchUIData] = synchronized {
  waitingBatchUIData.values.toSeq
}

def runningBatches: Seq[BatchUIData] = synchronized {
  runningBatchUIData.values.toSeq
```

```
}

def retainedCompletedBatches: Seq[BatchUIData] = synchronized {
  completedBatchUIData.toSeq
}
```

这3个函数分别返回等待的批次、正在运行的批次、已完成的批次的BatchUIData信息的序列。

回过头看源码3-159的StreamingPage.render，其中调用了generateBatchListTables。这3个函数被使用。

源码3-164 StreamingPage.generateBatchListTables

```
private def generateBatchListTables(): Seq[Node] = {
  val runningBatches = listener.runningBatches.sortBy(_.batchTime.milliseconds).reverse
  val waitingBatches = listener.waitingBatches.sortBy(_.batchTime.milliseconds).reverse
  val completedBatches = listener.retainedCompletedBatches.
    sortBy(_.batchTime.milliseconds).reverse

  val activeBatchesContent = {
    <h4 id="active">Active Batches ({runningBatches.size + waitingBatches.size})</h4> ++
      new ActiveBatchTable(runningBatches, waitingBatches, listener.batchDuration).toNodeSeq
  }

  val completedBatchesContent = {
    <h4 id="completed">
```

```
      Completed Batches (last {completedBatches.size} out of {listener.numTotalCompletedBatches})
    </h4> ++
      new CompletedBatchTable(completedBatches, listener.batchDuration).toNodeSeq
  }

  activeBatchesContent ++ completedBatchesContent
}
```

给变量runningBatches、waitingBatches、completedBatches赋值时分别调用了这3个函数，而接着更新StreamingPage页面信息时用到了这3个变量，分别用于显示正在运行的批次、等待的批次、已完成的批次的信息。

3.12 Spark Streaming应用程序的停止

一般的Spark Streaming应用程序在正常运行情况下是不用停止的，因为流处理能始终处于运转状态。但在某些复杂应用程序中或者遇到异常时，是需要停止应用程序的。

Spark Streaming应用程序中，可利用StreamingContext.stop来停止应用程序。在使用StreamingContext的start和stop时，需要注意以下几点：

（1）一旦StreamingContext启动，就不能再对其计算逻辑进行添加或修改。

（2）一旦StreamingContext被停止，就不能重启。

（3）单个JVM虚拟机同一时间只能包含一个active的StreamingContext。

（4）StreamingContext.stop也会把关联的SparkContext对象停止，如果不想把SparkContext对象也停止，可以将StreamingContext.stop的可选参数 stopSparkContext 设

为false。

（5）一个SparkContext对象可以和多个StreamingContext对象关联，只要先对前一个StreamingContext.stop(stopSparkContext=false)，然后再创建新的StreamingContext对象即可。

下面分析代码。先看StreamingContext.stop。

源码3-165 StreamingContext.stop

```
def stop(
    stopSparkContext: Boolean = conf.getBoolean("spark.streaming.stopSparkContextByDefault", true)
  ): Unit = synchronized {
  stop(stopSparkContext, false)
}
```

在上面的代码中调用了另一个stop方法。

源码3-166 StreamingContext.stop

```
def stop(stopSparkContext: Boolean, stopGracefully: Boolean): Unit = {
  var shutdownHookRefToRemove: AnyRef = null
  if (AsynchronousListenerBus.withinListenerThread.value) {
    throw new SparkException("Cannot stop StreamingContext within listener thread of" +
      " AsynchronousListenerBus")
  }
  synchronized {
    try {
      state match {
```

```
      case INITIALIZED =>
        logWarning("StreamingContext has not been started yet")
      case STOPPED =>
        logWarning("StreamingContext has already been stopped")
      case ACTIVE =>
        scheduler.stop(stopGracefully)
        // Removing the streamingSource to de-register the metrics on stop()
        env.metricsSystem.removeSource(streamingSource)
        uiTab.foreach(_.detach())
        StreamingContext.setActiveContext(null)
        waiter.notifyStop()
        if (shutdownHookRef != null) {
          shutdownHookRefToRemove = shutdownHookRef
          shutdownHookRef = null
        }
        logInfo("StreamingContext stopped successfully")
    }
  } finally {
    // The state should always be Stopped after calling `stop()`, even if we haven't started yet
    state = STOPPED
  }
}
if (shutdownHookRefToRemove != null) {
  ShutdownHookManager.removeShutdownHook(shutdownHookRefToRemove)
}
```

```
  // Even if we have already stopped, we still need to attempt to stop the SparkContext because
  // a user might stop(stopSparkContext = false) and then call stop(stopSparkContext = true).
  if (stopSparkContext) sc.stop()
}
```

把JobScheduler、StreamingSource、StreamingTab、ContextWaiter等类型的应用程序相关的对象停止并注销，应用程序才得以停止。

stopGracefully参数默认是false，生产环境应该设置为 true，具体做法是，在配置文件中把spark.streaming.stopGracefullyOnShutdown设置为true，这样能让已运行的程序运行完再停止，以保证数据处理的完整。

如果Spark Core由于某些原因要停止运行，Spark Streaming程序是怎么做到停止的呢？这就会用到StreamingContext.stopOnShutDown和Spark Core中的ShutdownHookManager。

StreamingContext.stopOnShutDown调用了上面的stop。

源码3-167 StreamingContext.stopOnShutDown

```
private def stopOnShutdown(): Unit = {
  val stopGracefully = conf.getBoolean("spark.streaming.stopGracefullyOnShutdown", false)
  logInfo(s"Invoking stop(stopGracefully=$stopGracefully) from shutdown hook")
  // Do not stop SparkContext, let its own shutdown hook stop it
  stop(stopSparkContext = false, stopGracefully = stopGracefully)
}
```

在StreamingContext.start中，会把stopOnShutDown的hook加入ShutdownHookManager。

源码3-168　StreamingContext.start

```
def start(): Unit = synchronized {
  state match {
    case INITIALIZED =>
      startSite.set(DStream.getCreationSite())
      StreamingContext.ACTIVATION_LOCK.synchronized {
        StreamingContext.assertNoOtherContextIsActive()
        try {
          validate()

          // Start the streaming scheduler in a new thread, so that thread local properties
          // like call sites and job groups can be reset without affecting those of the
          // current thread.
          ThreadUtils.runInNewThread("streaming-start") {
            sparkContext.setCallSite(startSite.get)
            sparkContext.clearJobGroup()
            sparkContext.setLocalProperty(SparkContext.SPARK_JOB_INTERRUPT_ON_CANCEL, "false")
            scheduler.start()
          }
          state = StreamingContextState.ACTIVE
        } catch {
          case NonFatal(e) =>
            logError("Error starting the context, marking it as stopped", e)
            scheduler.stop(false)
            state = StreamingContextState.STOPPED
```

```
            throw e
        }
        StreamingContext.setActiveContext(this)
      }
      shutdownHookRef = ShutdownHookManager.addShutdownHook(
        StreamingContext.SHUTDOWN_HOOK_PRIORITY)(stopOnShutdown)
      // Registering Streaming Metrics at the start of the StreamingContext
        assert(env.metricsSystem != null)
      env.metricsSystem.registerSource(streamingSource)
      uiTab.foreach(_.attach())
      logInfo("StreamingContext started")
    case ACTIVE =>
      logWarning("StreamingContext has already been started")
    case STOPPED =>
      throw new IllegalStateException("StreamingContext has already been stopped")
  }
}
```

所以，在StreamingContext启动时，就已经利用ShutdownHookManager增加了一个用于停止StreamingContext的钩子。ShutdownHookManager是Spark Core中定义的用于管理停止钩子的类。下面看看其中的部分代码。

源码3-169 ShutdownHookManager片段

```
/**
 * Various utility methods used by Spark.
```

```
 */
private[spark] object ShutdownHookManager extends Logging {
  val DEFAULT_SHUTDOWN_PRIORITY = 100

  /**
   * The shutdown priority of the SparkContext instance. This is lower than the default
   * priority, so that by default hooks are run before the context is shut down.
   */
  val SPARK_CONTEXT_SHUTDOWN_PRIORITY = 50

  /**
   * The shutdown priority of temp directory must be lower than the SparkContext shutdown
   * priority. Otherwise cleaning the temp directories while Spark jobs are running can
   * throw undesirable errors at the time of shutdown.
   */
  val TEMP_DIR_SHUTDOWN_PRIORITY = 25

  private lazy val shutdownHooks = {
    val manager = new SparkShutdownHookManager()
    manager.install()
    manager
  }
  ...
  // 增加钩子，就是把有优先级的函数加入队列中
  def addShutdownHook(priority: Int)(hook: () => Unit): AnyRef = {
    shutdownHooks.add(priority, hook)
```

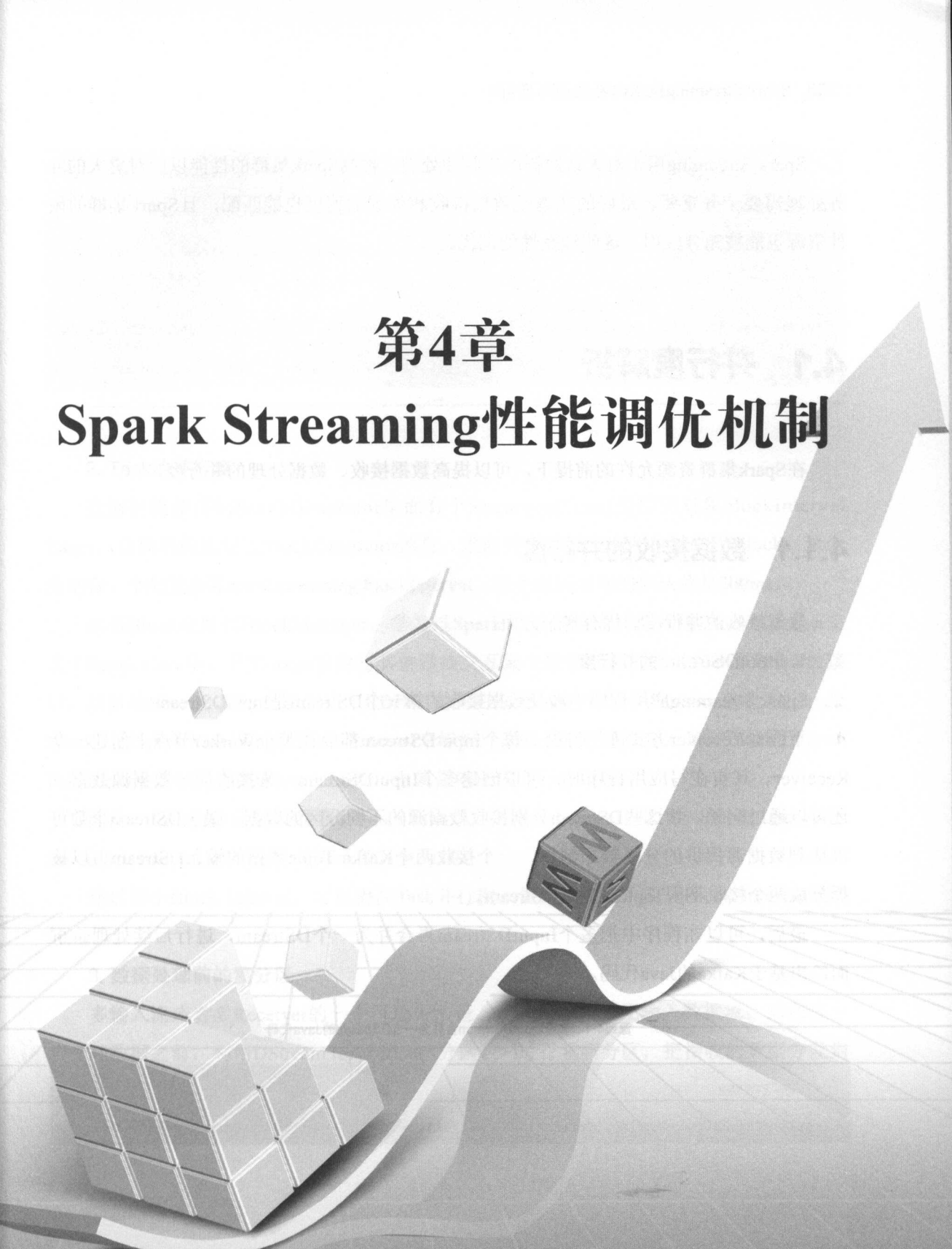
```
  }
  ...
}

private [util] class SparkShutdownHookManager {

  // 优先级队列，优先级越大，越优先执行
  private val hooks = new PriorityQueue[SparkShutdownHook]()
  @volatile private var shuttingDown = false

  /**
   * Install a hook to run at shutdown and run all registered hooks in order. Hadoop 1.x does not
   * have `ShutdownHookManager`, so in that case we just use the JVM's `Runtime` object and hope for
   * the best.
   */
  // 实例化一个线程，添加到JVM的关闭钩子中，等到JVM退出时才会被调用
  def install(): Unit = {
    val hookTask = new Runnable() {
      override def run(): Unit = runAll()
    }
    Try(Utils.classForName("org.apache.hadoop.util.ShutdownHookManager")) match {
      case Success(shmClass) =>
        val fsPriority = classOf[FileSystem]
          .getField("SHUTDOWN_HOOK_PRIORITY")
          .get(null) // static field, the value is not used
          .asInstanceOf[Int]
```

```
        val shm = shmClass.getMethod("get").invoke(null)
        shm.getClass().getMethod("addShutdownHook", classOf[Runnable], classOf[Int])
          .invoke(shm, hookTask, Integer.valueOf(fsPriority + 30))

      case Failure(_) =>
        Runtime.getRuntime.addShutdownHook(new Thread(hookTask, "Spark Shutdown Hook"));
    }
  }

  // JVM退出时钩子回调此函数
  def runAll(): Unit = {
    shuttingDown = true
    var nextHook: SparkShutdownHook = null
    while ({ nextHook = hooks.synchronized { hooks.poll() }; nextHook != null }) {
      Try(Utils.logUncaughtExceptions(nextHook.run()))
    }
  }

  def add(priority: Int, hook: () => Unit): AnyRef = {
    hooks.synchronized {
      if (shuttingDown) {
        throw new IllegalStateException("Shutdown hooks cannot be modified during shutdown.")
      }
      val hookRef = new SparkShutdownHook(priority, hook)
      hooks.add(hookRef)
      hookRef
```

```
    }
  }

  def remove(ref: AnyRef): Boolean = {
    hooks.synchronized { hooks.remove(ref) }
  }

}

private class SparkShutdownHook(private val priority: Int, hook: () => Unit)
  extends Comparable[SparkShutdownHook] {

  override def compareTo(other: SparkShutdownHook): Int = {
    other.priority - priority
  }

  def run(): Unit = hook()

}
```

所以StreamingContext启动时调用ShutdownHookManager.addShutdownHook，就是把默认优先级为51的stopOnShutdown函数放入SparkShutdownHookManager的优先级队列hooks中。JVM退出时，启动一个线程，调用SparkShutdownHookManager.runAll方法，即从hooks队列中按优先级的顺序取出并执行其中的函数。而stopOnShutdown函数在队列中，所以调用了stopOnShutdown函数，从而调用StreamingContext.stop函数，应用程序就可以执行停止工作了。

第4章
Spark Streaming性能调优机制



Spark Streaming用于对大量数据的接收和处理，提高Spark集群的性能以应对更大的业务处理需要十分重要。最好的状态是数据接收和处理的速度也能匹配，且Spark集群的硬件资源也能被充分应用。这就涉及性能调优。

4.1 并行度解析

在Spark集群资源允许的前提下，可以提高数据接收、数据处理的并行度。

4.1.1 数据接收的并行度

数据接收的并行度调优有多个方面。

1. InputDStream的并行度

Spark Streaming应用程序中涉及数据接收的第一个DStream是InputDStream。

下面对Receiver方式进行讨论。每个InputDStream都会在某个Worker节点上创建一个Receiver。其实在写应用程序时，可以创建多个InputDStream，来接收同一数据源数据。还可以通过配置，让这些DStream分别接收数据源的不同分区的数据，最大DStream个数可以达到数据源提供的分区数。例如，一个接收两个Kafka Topic数据的输入DStream可以被拆分成两个接收不同Topic数据的DStream。

最后，可以在程序中把多个InputDStream再合并为一个DStream，进行后续处理。下面给出基于Kafka的Java代码。

源码4-1 多个InputDStream合并为一个DStream的Java代码

```
int numStreams = 5;

List<JavaPairDStream<String, String>> kafkaStreams = new ArrayList<JavaPairDStream<String,
```

```
 String>>(numStreams);
for (int i = 0; i < numStreams; i++) {
  kafkaStreams.add(KafkaUtils.createStream(...));
}
JavaPairDStream<String, String> unifiedStream = streamingContext.union(kafkaStreams.get(0),
kafkaStreams.subList(1, kafkaStreams.size()));
```

2. Task的并行度

数据接收使用的BlockGenerator里面有个RecurringTimer类型的对象blockIntervalTimer，会周期性地发送BlockGenerator消息，进而周期性地生成和存储一个Block。这个周期有一个配置参数spark.streaming.blockInterval。这个时间周期的默认值是200ms。

读写Block会用到BlockManager。学习过Spark Core的读者，会知道BlockManager定义于Spark Core中，是Storage模块与其他模块交互最主要的类，提供了读和写Block的接口。这里的Block，实际上就对应了RDD中提到的Partition，每一个Partition都会对应一个Block。而Spark Streaming按Batch Interval来组织一次数据接收和处理，所以Batch Interval内的Block个数就是RDD的Partition数，也就是RDD的并行Task数。

因此，Task的并行度大致等于Batch Interval / Block Interval。比如，Batch Interval是2s，Block Interval是200ms，则Task并行度为10。

通过调小Block Interval，可以提高Task并行度。但建议一般不要让Block Interval低于50ms。

3. 数据处理前的重分区

多输入流或者多Receiver的一个可选方法是，明确地重新分配输入数据流，即在进一步处理数据之前，利用DStream.repartition(<分区数>)进行重新分区，把接收的数据分发到集群上。

4.1.2 数据处理的并行度

如果运行在计算stage上的并发任务数不是足够大，就不会充分利用集群的资源。例如，对于分布式reduce操作，如reduceByKey和reduceByKeyAndWindow，默认的并发任务数通过配置属性spark.default.parallelism来确定。可以通过参数PairDStreamFunctions传递并行度，或者设置参数 spark.default.parallelism修改默认值。

4.2 内存

Spark Streaming应用需要的集群内存资源是由使用的转换操作类型决定的。举例来说，如果想要使用一个窗口长度为10分钟的窗口操作，那么集群就必须有足够的内存来保存10分钟内的数据。如果想要使用updateStateByKey来维护许多key的state，那么内存资源就必须足够大。反过来说，如果想要做一个简单的map-filter-store操作，那么需要使用的内存就很少。

通常来说，通过Receiver接收到的数据会使用StorageLevel.MEMORY_AND_DISK_SER_2持久化级别来进行存储，因此无法保存在内存中的数据会溢写到磁盘上，而溢写到磁盘上会降低应用的性能。因此，通常是建议为应用提供它需要的足够的内存资源。建议在一个小规模的场景下测试内存的使用量，并进行评估。

4.3 序列化

Spark Streaming默认将接收到的数据序列化存储，以减少内存使用。序列化和反序列

化需要更多的CPU时间，更加高效的序列化方式（Kryo）和自定义的序列化接口可以更高效地使用CPU。使用Kryo时，一定要考虑注册自定义的类，并且禁用对应引用的tracking（spark.kryo.referenceTracking）。

在流式计算的场景下，有两种类型的数据需要序列化。

（1）输入数据。默认情况下，接收到的输入数据是存储在Executor的内存中的，使用的持久化级别是StorageLevel.MEMORY_AND_DISK_SER_2。这意味着，数据被序列化为字节从而减小GC开销，并且会复制以进行Executor失败的容错。因此，数据首先会存储在内存中，然后在内存不足时会溢写到磁盘上，从而为流式计算来保存所有需要的数据。这里的序列化有明显的性能开销：Receiver必须反序列化从网络接收到的数据，然后再使用Spark的序列化格式序列化数据。

（2）流式计算操作生成的持久化RDD。流式计算操作生成的持久化RDD可能会持久化到内存中。例如，窗口操作默认会将数据持久化在内存中，因为这些数据后面可能会在多个窗口中被使用，并被处理多次。然而，不像Spark Core的默认持久化级别StorageLevel.MEMORY_ONLY，DStream的默认持久化级别是MEMORY_ONLY_SER，默认就会减小GC开销。

在一些特殊的场景中，比如需要为流式应用保持的数据总量并不是很多时，也许可以将数据以非序列化的方式进行持久化，从而减少序列化和反序列化的CPU开销，而且又不会有太昂贵的GC开销。举例来说，如果是数秒的batch interval，并且没有使用window操作，那么可以考虑通过显式地设置持久化级别来禁止持久化时对数据进行序列化。这样就可以减少用于序列化和反序列化的CPU性能开销，并且不用承担太多的GC开销。

4.4 Batch Interval

要确保Spark Streaming应用程序在集群环境下稳定运行，系统必须尽快处理接收的数

据。处理数据的速度要跟上数据流入的速度，即批次处理时间必须小于批次间隔时间。通过查看日志可以了解Total delay，如果delay小于Batch Interval，则系统稳定。

如果Delay一直增加，则说明系统的处理速度跟不上数据输入速度，要做调整。要想有一个稳定的配置，可以尝试提升数据处理的速度，或者增加Batch Interval。

临时性的数据增长导致的暂时的延迟增长，可以认为是合理的，只要延迟情况可以在短时间内恢复即可。

4.5 Task

任务的提交和分发都会有延迟，需要注意Batch Interval不能太小。可采取以下措施：

（1）任务Kryo序列化可以减少任务大小，减少发送到Worker节点的时间。

（2）Standalone模式、粗粒度Mesos模式下的Spark比细粒度模式有更低的延迟。

4.6 JVM GC

GC会影响任务的运行。应采取不同策略以减小GC对Job运行的影响。减低系统吞吐量就能减小GC的停顿。

对于流式应用来说，如果要获得低延迟，肯定不希望出现因为JVM垃圾回收导致的长时间延迟。有很多参数可以帮助降低内存使用和GC开销：

（1）DStream的持久化。输入数据和某些操作生产的中间RDD，默认持久化时都会序列化为字节。与非序列化的方式相比，这会降低内存和GC开销。使用Kryo序列化机制可以进一步减少内存使用和GC开销。进一步降低内存使用率，可以对数据进行压缩，由

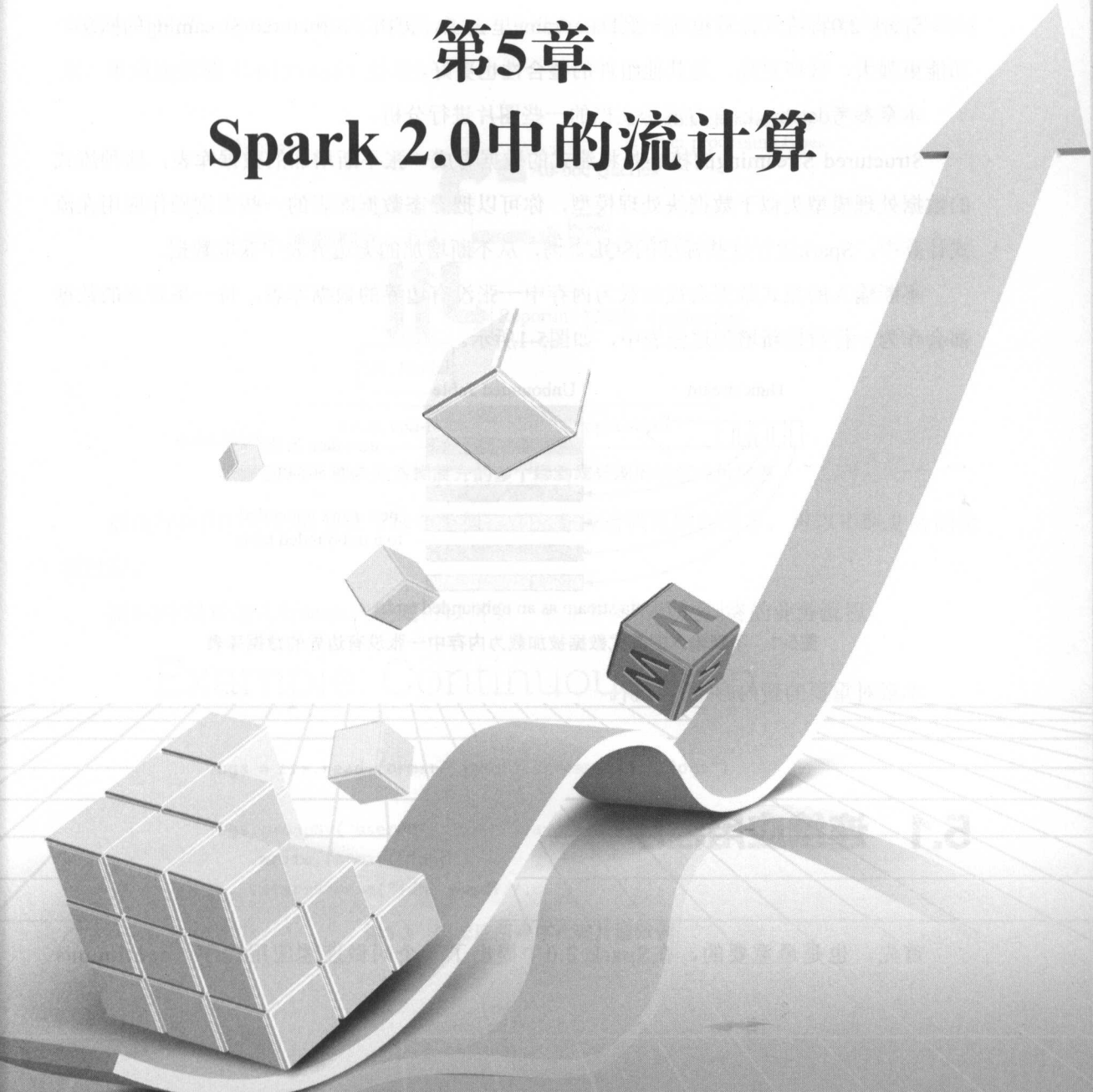
spark.rdd.compress参数控制（默认false）。

（2）清理旧数据。默认情况下，所有输入数据和通过DStream的转换操作生成的持久化RDD会自动被清理。Spark Streaming会决定何时清理这些数据，取决于转换操作类型。例如，在使用窗口长度为10分钟内的window操作，Spark会保持10分钟以内的数据，时间过了以后就会清理旧数据。但是在某些特殊场景下，比如Spark SQL和Spark Streaming整合使用时，在异步开启的线程中，使用Spark SQL针对Batch RDD进行执行查询。那么就需要让Spark保存更长时间的数据，直到Spark SQL查询结束。可以使用StreamingContext.remember方法来实现。

（3）CMS垃圾回收器。使用并行的mark-sweep垃圾回收机制。推荐使用该参数以保持GC低开销。虽然并行的GC会降低吞吐量，但是还是建议使用它来减少Batch的处理时间（降低处理过程中的GC开销）。如果要使用该机制，那么要在Driver端和Executor端都开启。具体做法是用spark-submit提交应用程序执行时增加两个设置：

```
--driver-java-options "-XX:+UseConcMarkSweepGC"
--conf "spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepGC"
```

第5章
Spark 2.0中的流计算



本书是针对Spark 1.6.1中的Spark Streaming。后来的Spark 1.6.2只是对少量bug做了修补。Spark 2.0之前的版本没有什么变化。

令人瞩目的Spark 2.0已出来，变化不小，但仍有bug，还不适合在生产环境下使用。

Spark 2.0版本更新包含3个主题：Easier、Faster、Smarter。

Spark 2.0将流式计算也统一到DataFrame里去了，提出了Structured Streaming的概念。功能更强大，效率更高，与其他组件的整合性也更好。

本章参考databricks官方演示文档的一些图片进行分析。

Structured Streaming的核心是将流式的数据看成一张不断增加的数据库表，这种流式的数据处理模型类似于数据块处理模型，你可以把静态数据库表的一些查询操作应用在流式计算中，Spark运行这些标准的SQL查询，从不断增加的无边界表中获取数据。

不断输入的流式数据会被加载为内存中一张没有边界的数据库表，每一条新来的数据都会作为一行数据新增到这张表中，如图5-1所示。

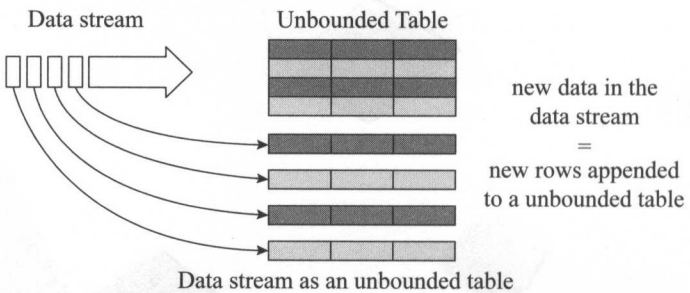


图5-1 不断输入的流式数据被加载为内存中一张没有边界的数据库表

本章对重要的新内容进行分析。

5.1 连续应用程序

首先，也是最重要的，在Spark 2.0中提出了一个叫做连续应用程序（continuous

application）的概念。

图5-2展示了一个围绕流数据展开的各种业务。数据从Kafka中流进来，通过ETL操作进行数据清洗，清洗出来作为目标数据进行进一步处理，可能是机器学习，也可能是交互式查询，也有可能直接把数据存在数据库或者其他外部存储设备，还有可能是直接交给已有的应用程序。Spark 2.0把流数据看成是一个没有边际的表，并能把全部处理环节串联起来，形成端到端（end to end）处理。

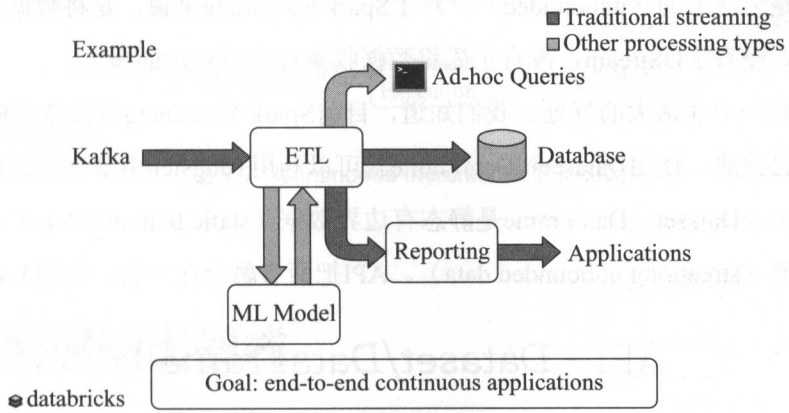


图5-2　围绕流数据展开的多个端到端处理的连续应用程序

而连续应用程序的模型就与这个类似，在充分应对风险的前提下，可以串联业务的全部过程。

图5-3中简短的几行scala代码就可以贯穿一个业务案例从始到终的业务流程。

Example: Continuous App

```
logs = ctx.read.format("json").stream("s3://logs")

logs.groupBy("userid", "hour").avg("latency")
    .write.format("jdbc")
    .startStream("jdbc:mysql//...")
```

图5-3　连续应用程序代码片段示例

5.2 无边界表unbounded table

对Spark Streaming来说连续（continuous）还有另一层含义，即运行在Dataset和DataFrame之上。

基本观点是把数据看成一张表，默认情况下Dataset和DataFrame中的表是有边界的，而在流处理中是无边界的（unbounded）。对于Spark Streaming来说，是将数据抽象为一个没有边界的表。没有了DStream，没有了先将数据收集过来再处理的概念。

这个做法有一个非常大的好处。我们知道，目前Spark Streaming直接依赖RDD，优化需要开发者自己完成，使用Dataset和DataFrame就可以利用Tungsten引擎来进行优化。

默认情况下，Dataset、DataFrame是静态有边界数据（static bounded data），流数据是流式无边界数据（streaming unbounded data）。API把两者融合在一起，如图5-4所示。

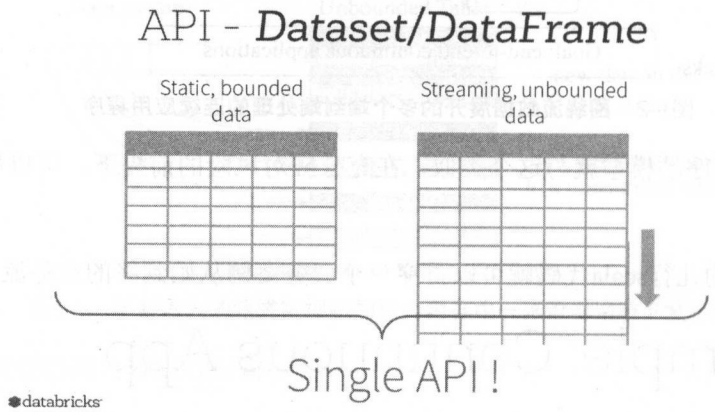


图5-4 Dataset/DataFrame API

如图5-5所示，新加入的Planner就类似路由器，我们在使用时，可以按照时间说明，由Planner确定每次读取的位置，在运行时动态绑定位置。在这种模式下，没有数据收集再处理的概念，可以认为数据一直在那儿，直接拿了处理就行。这可以极大地简化对流处理的理解。

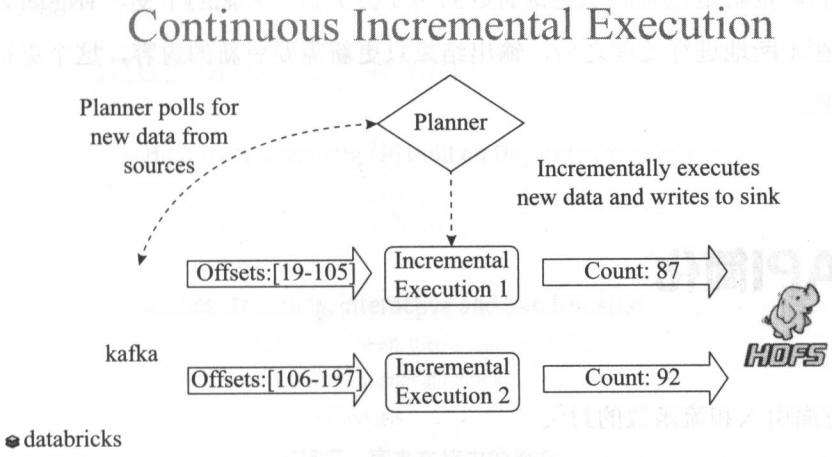


图5-5　Continuous Incremental Execution

5.3　增量输出模式

在Spark 2.x中，增加了多个输出模式，增量输出（delta output）是其中最重要的一种，如图5-6所示。

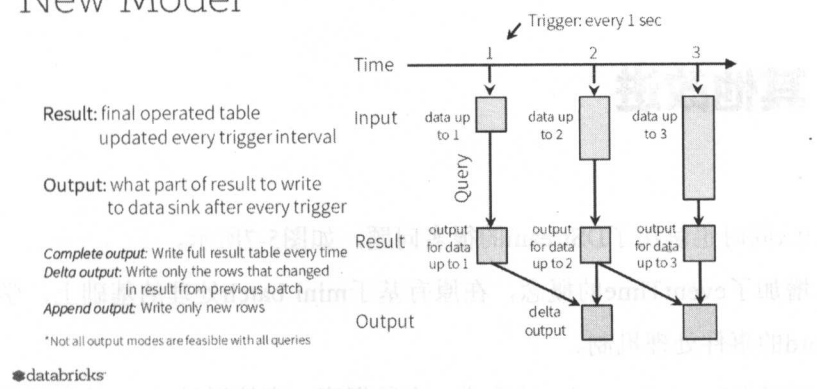


图5-6　增量输出模式

增量更新，也就是说有需要更新的数据的才会更新，其他的不变。Trigger会不断检测输入数据，在不断地进行处理之后，输出结果只更新需要更新的内容，这个更符合应用程序的处理场景。

5.4 API简化

在API方面引入和流函数的封装。

这里举个例子：Kafka中读取的数据，通过stream方法形成流，就可以直接与JDBC中读取的数据在Dataset层面进行join，不用使用transform或者foreachRDD方法。

```
kafkaDataset = spark.read.akfka("iot-updates").stream()

staticDataset = ctxt.read.jdbc("jdbc://", "iot-device-info")

joinedDataset = kafkaDataset.join(staticDataset, "device-type")
```

stream方法底层依赖Dataset和DataFrame，集成了Spark SQL和Dataset几乎所有的功能，把流处理的代码编写一下简化了很多。

5.5 其他改进

Spark 2.x同时也解决了DStream的很多问题，如图5-7所示。

（1）增加了eventTime的概念，在原有基于mini batch处理的基础上，学习了Storm基于每个record的事件处理机制。

（2）可以把Spark Streaming抽象成一个数据库，直接通过JDBC访问数据。

（3）在运行时可以变更query，并支持多个query并行运行。

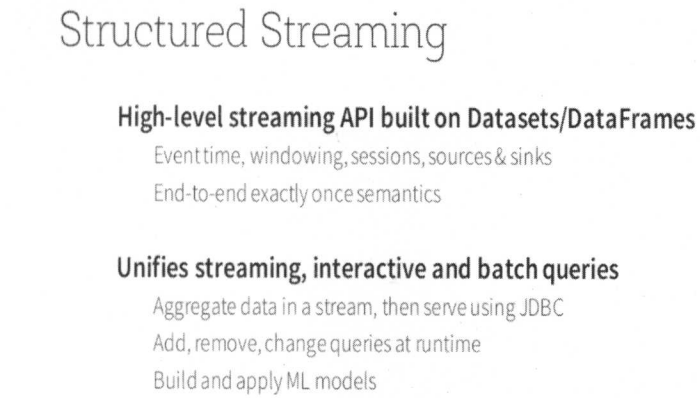


图5-7　官方文档中的原文

总之，从Spark 2.x的设计来看，从根本上，是为了满足更快、完全容错、完全的语义一致性exactly-once的要求。通过实现有状态流处理，让应用程序的功能更强大。而基于Dataset和DataFrame处理，让我们忘记流的概念，使用将会越来越简单。

技术内幕及源码剖析

全面透彻剖析Spark Streaming技术内幕和源码，并结合Spark Streaming调优实践的经验,涵盖Spark Streaming的内部技术原理、源码分析、性能调优方法以及对未来Spark Streaming新版本的新功能分析，特别适合有志于成为大数据高手的朋友阅读。

对大数据感兴趣的读者，
可关注DT大数据梦工厂
微信公众号

销售分类：计算机/大数据技术

ISBN 978-7-302-46491-4

9 787302 464914 >

定价：49.00元

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com